Institut für Parallele und Verteilte Systeme

Abteilung Simulation großer Systeme

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Master's Thesis Nr. 47

Design and Implementation of a Fault Simulation Layer for the Combination Technique on HPC Systems

Johannes Walter

Studiengang:	Informatik
Prüfer:	Jun.-Prof. Dr. rer. nat. Dirk Pflüger
Betreuer:	M.Sc. Mario Heene
begonnen am:	13.07.2015
beendet am:	12.01.2016
CR-Klassifikation:	D.4.1, D.4.4, D.4.5, D.4.7, D.4.8

Zusammenfassung

In heutigen Supercomputern wird hohe Rechenleistung durch eine große Anzahl an parallel betriebenen Prozessoren erzielt. Mit wachsender Anzahl gleichzeitig benutzter Prozessoren erhöht sich jedoch die Wahrscheinlichkeit für das Auftreten von Hardwarefehlern und daraus resultierten Prozessabstürzen. Ein weitverbreiteter Standard zum Austausch von Nachrichten in Netzwerken ist MPI. Aktuelle MPI-Versionen sind nicht fehlertolerant und terminieren im Falle von Fehlern das ganze MPI-Netzwerk. ULFM, eine fehlertolerante Erweiterung für MPI, ist nicht stabil implementiert und ist auf Supercomputern nicht verfügbar.
In dieser Masterarbeit wird ein Konzept für einen Fehlersimulator als Zwischenschicht zwischen MPI und der Anwendung vorgestellt und implementiert, mit dessen Hilfe Prozessabstürze und das Verhalten von ULFM simuliert werden können, ohne dass das darunterliegende MPI Netzwerk terminiert wird.

Abstract

In today's supercomputers, computing power is achieved by using a large amount of parallel executed processors. With growing amount of simultaneously used processors, the probability of hardware faults with resulting process failures grows as well. A popular standard for exchanging messages in networks is MPI. Current MPI versions are not fault-tolerant and terminate the whole MPI network in case of faults. ULFM, which is a proposed fault-tolerant extension of MPI, is not stable implemented and not available on supercomputers.
In this master's thesis, a concept of a fault simulator as intermediate layer between MPI and application is introduced and implemented. By means of this fault simulator, process crashes and the behavior of ULFM shall be able to be simulated, without resulting in termination of the underlying MPI network.

Contents

1 Introduction and basics

1.1 Introduction

MPI (section 1.3) is used in supercomputers to transfer messages and access distributed stored data. With growing network size the probability of process failures due to hardware malfunction is rising steadily. The current MPI standard does not support handling process faults. After a failure is detected, the MPI network is terminated. A proposed standard ULFM (section 2.3), an addition to the current MPI standard, offers fault tolerance. A prototype implementation of ULFM exists (implemented in Open MPI), but supercomputers mostly use custom implementations of MPI and thus ULFM is not available. There are numerical methods like

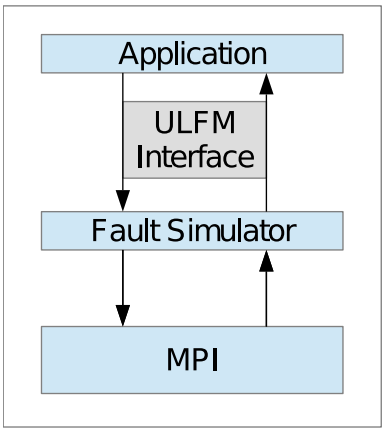


Figure 1.1: Fault simulator as layer between MPI and the application.

the combination technique (chapter 3) , that can handle process faults. To be able to run and test these methods, a fault simulator is needed, providing the ability to simulate process failures.

In this thesis we discuss and create a fault simulator (chapter 5) as a layer between the regular MPI implementation and the application. The simulator shall support all important MPI functions used by the combi technique framework (chapter 4), like for example blocking and non-blocking point-to-point operations, or specific blocking collective operations. Also for the sake of realism the fault simulator should use as least resources as possible. In chapter 6.2 we define and execute some tests and discuss the test results. For the implementation the programming language C++ 11 is used. The C++ code located on the CD-R on the last page is part of this thesis. While the implementation is theoretically portable and could

be used on multiple operating systems, it is only tested on current Linux distributions, because Linux is widely used for high performance computing. The MPI environments used for testing the fault simulator were MPICH, Open MPI and the implementation on Hazel Hen.

In this thesis the information about MPI and its functions is directly taken from the MPI standard specification [MPI15], whereas the information regarding ULFM is taken from the proposed ULFM standard [ULF].

1.2 Definitions and terminology

In this section we define a few notions we use frequently. At first let's take a look at the definition of a deadlock according to [Tan08].

Definition 1.2.1 (Deadlock)
A deadlock is a state in a distributed system, where a set of processes wait for an event to occur, that can only be triggered by a process in the same set.

For example if we have two processes, p_1 and p_2. We have a deadlock if p_1 waits for a message from p_2, while p_2 waits for a message from p_1. In our fault layer we have to be really careful with deadlocks.

In our fault layer we use various distributed algorithms and protocols. We can think of them as a 'distributed function', where we input some values and expect a specific output. The following definitions of strong and weak consistency are based on [KRS+13].

Definition 1.2.2 (Strong consistency)
A distributed algorithm is strong consistent, if it behaves as if it was executed locally and any intermediate (inconsistent) steps are not visible to the application.

While strong consistency is very desirable, it is not possible to achieve it in every situation. Additionally it comes with a possible high overhead, because processes have to be synchronized in order to guarantee only consistent states.

Definition 1.2.3 (Weak consistency)
A distributed algorithm is weak consistent if it is possible to read inconsistent values within its execution, but eventually a consistent state is achieved.

An inconsistent state is for example if one process has access to the finished results of the algorithm, while another can access unfinished values.

The definitions of fault, failure and fault tolerance according to [IEE90].

Definition 1.2.4 ((Hardware) fault)
A defect in a hardware device or component; for example, a short circuit or broken wire.

Definition 1.2.5 (Failure)
The inability of a system or component to perform its required functions within specified performance requirements.

A fault might (but doesn't have to) lead to a failure. The cause of a failure might be as well hardware faults as also software faults (like software bugs).

Definition 1.2.6 (Fault tolerance)
The ability of a system or component to continue normal operation despite the presence of hardware or software faults.

1.3 MPI

MPI (Message Passing Interface, [MPI15]) is a standard for a portable message-passing interface in a network. Multiple implementations of MPI are available, like Open MPI or MPICH and its many derivatives (from vendors like IBM, Intel or Microsoft).
The current version of MPI is 3.1 (released 2015). It gives support for blocking and non-blocking point-to-point operations (such as MPI_SEND and the non-blocking version MPI_ISEND) and collective operations (for example MPI_BCAST and the non-blocking version MPI_IBCAST). Additionally MPI gives a standardized interface to access files, handle process topologies, dynamically spawn processes and more. Explicitly NOT included is support for interrupts or remote procedure calls.

While non-blocking point-to-point operations are included in the standard since MPI 1.0 (released 1994), non-blocking collective operations are included only since MPI 3.0 (released 2012) and thus possibly not included in all today used MPI implementations, especially regarding custom implementations in high performance computers. We therefore try to avoid using non-blocking collectives in our fault layer and use custom versions of MPI_IBCAST and MPI_IREDUCE (also see section 5.3.5.1).

2 Fault tolerance

(Also see chapter 8.3 in [MPI15])
The current MPI standard does not support handling process faults (especially process failures). If an error is detected by the MPI network, the MPI network will be aborted similar to executing MPI_ABORT and the execution state from this point will be undefined. It is possible to change the MPI error handler from MPI_ERRORS_ARE_FATAL to another handler like MPI_ERRORS_RETURN, which leads to the network not being terminated immediately after an error, but the state is still undefined and calling MPI functions after an error occurred might be not possible (depending on the implementation).

2.1 Fault types

Before we begin discussing an approach to simulate faults we first need to define, what is considered faulty behavior and which types of faults we intend to simulate.

Using the classifications and definitions from [KKL05], we have four types of fault models as depicted in Figure 2.1. Fail-Stop is a very common used model, where a process either works as specified or stops its execution completely. This might be detected by other processes. Fail-Stutter includes (detectable) hardware corruption like memory bit flips or performance loss, while the process might still continue its operation. Silent Fail-Stutter adds undetectable malfunction, like undetectable memory bit flips or performance loss due to hardware malfunction. Byzantine means the behavior is totally unspecified and anything could happen, including (detectable and undetectable) corrupted data, freezes, undetectable random message initiation in the network (concerning MPI), etc.

Byzantine is the most undesirable kind of faulty behavior, because although a faulty process acts totally undefined and unpredictable, from the outside it might not be detectable and might look like the process performs just like the other processes.

For our fault simulator, we use the Fail-Stop model, meaning a process either works as expected without any faults, or the process crashes completely. In our fault simulator if a process is dead, it stays dead until the application is terminated by calling MPI_FINALIZE or MPI_ABORT. Additionally we briefly discuss Silent faults in section 5.4 and implement the possibility to simulate random bit flips.

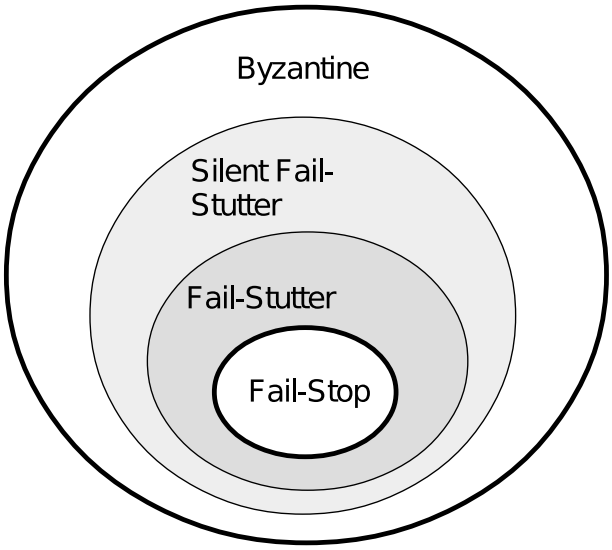


Figure 2.1: Different types of faults / fault models according to [KKL05].

2.2 Why fault tolerance?

We begin with a simple fictional example. Assume we have a supercomputer with 5000 nodes and the lifetime of a node (that means the time until it fails) is exponential distributed with an expected lifetime of $\frac{1}{\lambda} = 1000$ days. By further assuming, the lifetimes are stochastically independent, we can calculate the expected value of the event 'any node dies':

$$\mathbb{E}\left(\{\text{any node dies}\}\right) = \frac{1}{\sum_{k=1}^{5000} \frac{1}{1000}} = \frac{1000}{5000} = \frac{1}{5} \quad \text{(in days)}.$$

As you can see, while with one node a failure is expected after years, with 5000 nodes a failure is to be expected only after a few hours. Without any fault tolerance this would mean, the whole network fails, the failed component has to be repaired and the network has to be restarted every few hours.

One possibility to counteract this problem is to install redundant and fault tolerant components to reduce the risk of hardware failures or at least to reduce the impact. Another possibility is to make the software itself fault tolerant, this means for example instead of restarting the network after a failure is detected, the failed node is just excluded and the current operation is continued using only the other nodes.

2.3 ULFM

ULFM (User Level Fault Mitigation) [ULF] is a proposal of a fault tolerant MPI specification based on the (back then) upcoming MPI version 3.1. The proposal

is created by the MPI Forum's Fault Tolerance working group [1]. The idea is that instead of terminating the whole MPI network after an error is detected, the user has the possibility to catch errors and react to them without having to restart the MPI network.

2.3.1 New semantics

If no process failures occur, ULFM behaves just like the regular MPI. In case of a process failure, there are new MPI error codes that can be returned by point-to-point or collective operations to inform the application of a present failed process.
MPI_ERR_PROC_PENDING is returned for a non-blocking receive operation if its source is MPI_ANY_SOURCE and a possible matching sending process for the receive operation has failed.
MPI_ERR_PROC_FAILED is returned in the general case if an operation cannot be successfully completed because of a present process failure.

If a process failure is detected, the application can create new communicators excluding the failed processes and continue the execution. Process faults are only propagated to processes calling MPI operations involving the failed process. If for example a process wants to send a message to a failed process using MPI_SEND, only this process is notified of the failure.
In case the other processes in the communicator have to be informed about the dead process, the communicator can be revoked. As a result eventually all point-to-point and collective MPI operations in the revoked communicator return the error MPI_ERR_REVOKED.

2.3.2 New functions

We take a brief look at the ULFM functions implemented in our fault layer. For a complete and more detailed overview please take a look at the proposed ULFM standard [ULF].

int MPI_Comm_revoke(MPI_Comm comm)

Revokes a communicator and notifies all processes in the communicator. Eventually all operations (except MPI_COMM_SHRINK and MPI_COMM_AGREE) using the revoked communicator will be completed and return MPI_ERR_REVOKED.

int MPI_Comm_shrink(MPI_Comm comm, MPI_Comm* newcomm)

If process failures are detected, the dead processes can be excluded using this function. The newly created communicator 'newcomm' will contain all processes from 'comm' excluding the detected dead processes until this point.

[1] https://svn.mpi-forum.org/trac/mpi-forum-web/wiki/FaultToleranceWikiPage

int MPI_Comm_failure_ack(MPI_Comm comm)

This function can be used to acknowledge process failures. After this it is possible to receive a point-to-point message with source MPI_ANY_SOURCE (the application is responsible to make sure the sending process is not dead). Additionally the function MPI_COMM_FAILURE_GET_ACKED can be called to get a group containing all acknowledged dead processes in the communicator.

int MPI_Comm_failure_get_acked(MPI_Comm comm, MPI_Group* failedgrp)

Returns a group containing all acknowledged dead processes. Before calling this function, MPI_COMM_FAILURE_ACK has to be called first or otherwise the group will be empty.

int MPI_Comm_agree(MPI_Comm comm, **int*** flag)

If any process failure occurs in 'comm', this collective function will return the error MPI_ERR_PROC_FAILED at all alive processes in the communicator. If no process is dead, the function will return MPI_SUCCESS at all processes in the communicator.
Additionally this function performs a bitwise 'AND' operation with the values of flag from all participating alive processes. That means flag will be set to 0 at all processes if it is 0 at any alive process participating and it will stay 1 if flag is 1 at all alive processes.

3 Sparse grid combination technique

This chapter is based on [HKH+15] and [HHJP].

In section 2.2 we have discussed, why fault tolerance is important. With ULFM we have a standard that makes MPI fault tolerant. While ULFM offers the ability to detect and treat process faults without having to terminate the MPI network, the application has to be specifically programmed to react appropriately to dead processes.

In this chapter we have a brief look at the sparse grid combination technique, a numerical method that has the ability to efficiently handle detected process faults and thus gain significant advantage by using ULFM compared to restarting the whole computation or using checkpoint-restart.

The basic idea is for the discretization of a d-dimensional space $\Omega = [0,1]^d$, instead of using a uniform full grid Ω_n with mesh width $h_n = 2^{-n}$ requiring $\mathcal{O}(2^{nd})$ points, an approximation using a significantly lower amount of points is used, by increasing the approximation error only insignificantly. The approximation grid is created by linear combining smaller anisotropic grids Ω_l with mesh-width $h_{l_k} := 2^{-l_k}$ for the multi-index so called level-vector $l = [l_1, l_2, ..., l_d]$.

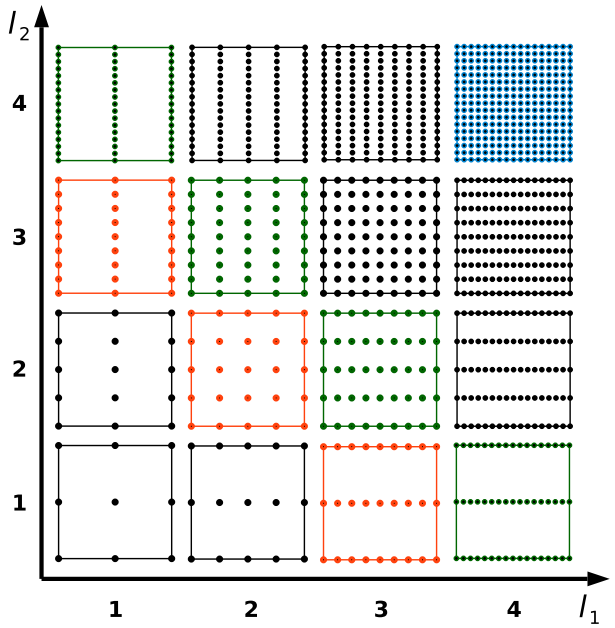


Figure 3.1: Full grid $\Omega_{[4,4]}$ (blue) gets approximated by combining the green grids (+) and the red grids (-).

Let $f_n(x)$ be the d-dimensional solution of a problem on the grid Ω_n. The idea is to approximate the solution by a weighted sum of solutions from smaller grids

$$f_n(x) \approx f_n^{(c)}(x) = \sum_{l \in \mathcal{L}} c_l f_l(x)$$

where $\mathcal{L}$ is a set of level vectors of grids Ω_l used for the combination and c_i are appropriate weights.

The special case

$$f_n^{(c)}(x) = \sum_{q=0}^{d-1} (-1)^q \binom{d-1}{q} \sum_{l \in \mathcal{L}_{n,q}} f_l(x)$$

is then the classic sparse grid combination technique with

$$\mathcal{L}_{n,q} = \{l \in \mathbb{N}^d : |l|_1 = |l_{\min}|_1 + \tau - q : l_{\min} \leq l \leq n\},$$

where $l_{\min} = n - \tau \mathbb{1}, \tau \in \mathbb{N}_0$, so that $l_{\min} \geq \mathbb{1}$.

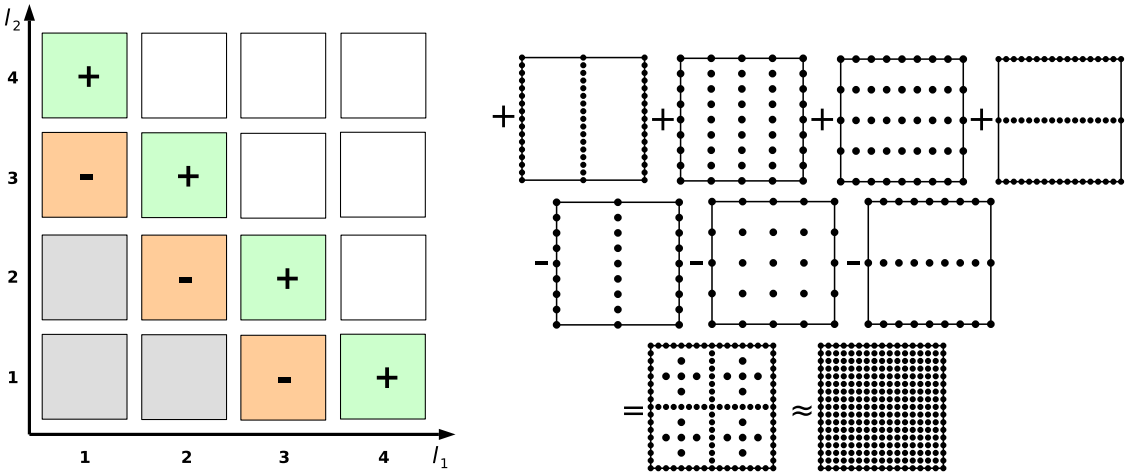


Figure 3.2: Combination technique for $d = 2$, $l_{\min} = [1, 1]$ and $n = [4, 4]$.

3.1 Handling process failures

If a process fails during calculation, instead of completely restarting the computation of the corresponding grid on another process, a failed calculation can be replaced by changing the weights of the combination technique and calculating smaller (and faster to calculate) problems, while increasing the approximation error only insignificantly. The choice of the weights is important, because bad grid replacements might increase the approximation error noticeable or cause unnecessary computation cost. For more information see for example [HKH+15].

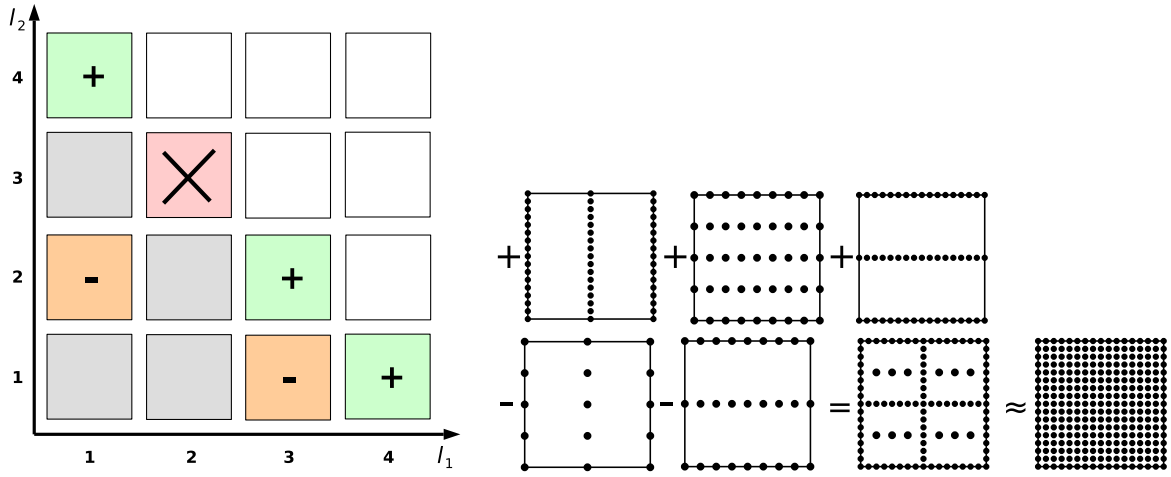


Figure 3.3: A possible combination of the last example with failed calculation of grid $\Omega_{[2,3]}$. Only the solution of grid $\Omega_{[1,2]}$ has to be calculated instead of $\Omega_{[2,3]}$.

4 Combi technique framework

The main purpose of this thesis is to be able to use the fault simulator in the combi technique framework. In this chapter we have a brief look at the construction of the framework.

4.1 Master/worker concept

In the framework we have numerous working groups consisting of a specific amount of worker processes and one master process. Additionally we have one single manager process, communicating with the master processes and sending them new jobs to compute (see Figure 4.1). The master processes receive the jobs and forward them to their worker processes. After the job computation is completed, the master process contacts the manager process and asks about the next job. The workers do not communicate with other processes outside their working group. If information needs to be exchanged with other groups, this is done merely by communication between the master processes and the manager process.

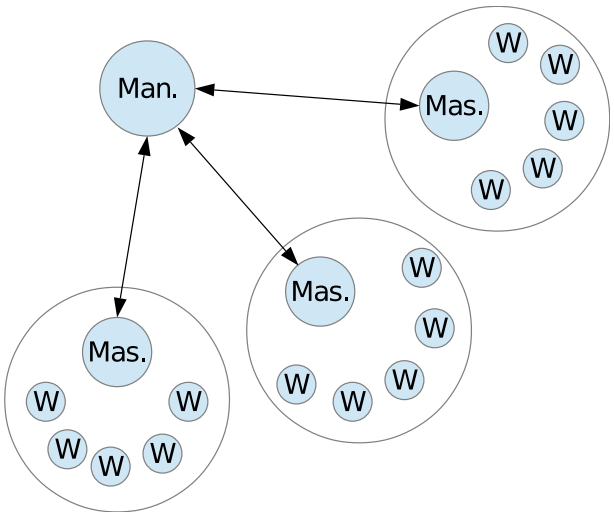


Figure 4.1: The Manager process distributes jobs to the master processes of each group. The master processes forward the jobs to their workers.

5 Fault simulator

5.1 Consistency and first ideas

While creating and developing the fault simulator, some initial approaches lead to problems concerning consistency and deadlocks. To have a better understanding about the thought process of creating the fault simulator, we take a look at an approach with possible deadlock (see Figure 5.1 as an initial example). The first

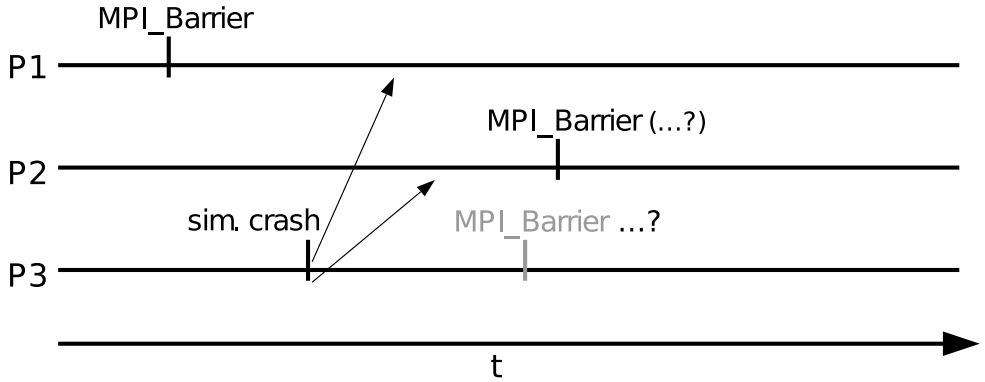
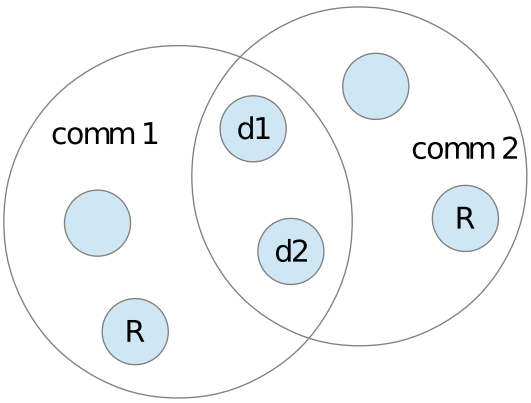


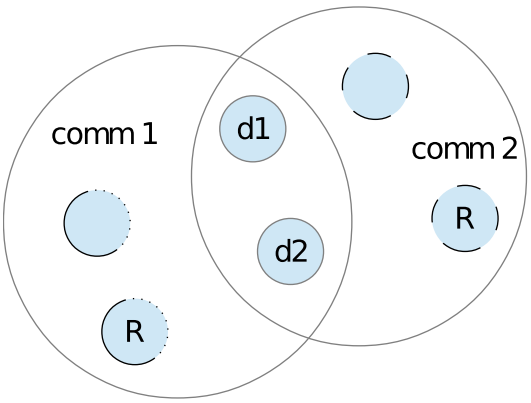
Figure 5.1: Process 3 dies, while process 1 already initiated MPI_Barrier. Possible dead lock if not treated right.

idea for blocking collectives was for every process to immediately execute blocking collectives after called by the application. The root process of every communicator would then have to inform possible dead processes about recently started blocking collectives, "command" them to participate in the blocking collective and unblock all processes. This approach was quite promising regarding performance, because while no process was dead, no further communication was necessary and the performance would have been equal to the regular MPI.
However, the root process had to know about all dead processes and send a command-message before its own blocking collective was executed, forcing all processes to send a message to root prior to any blocking collective operation. Furthermore some special cases like the example in Figure 5.2 lead to a deadlock. To be able to avoid this deadlock, a more complex treatment of process failures was required. This ultimately lead to the implemented version, where all process faults are synchronized before any "real" blocking MPI collective is executed.

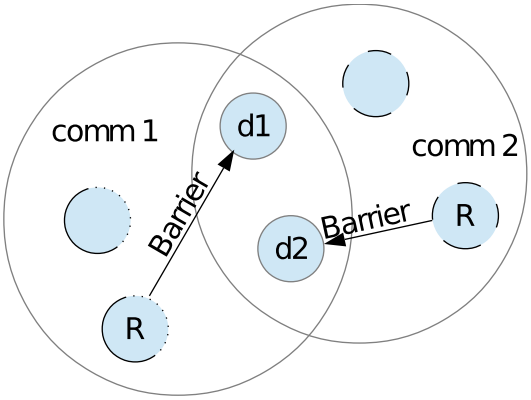
Further approaches are discussed in Section 5.5.



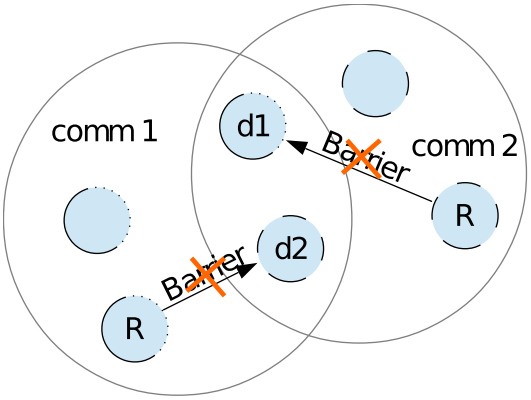
(a) Two communicators with dead processes d1 and d2 being member of both communicators.



(b) Processes of both communicators start a MPI_BCAST operation (indicated by the border). Note that the root processes technically start the barrier only after they sent their last point-to-point message (after (d)).



(c) Both root processes begin to command the dead processes to execute an MPI_BARRIER.



(d) d1 begins a barrier in comm1, d2 begins a barrier in comm2. Further command messages are not received. Deadlock.

Figure 5.2: Special case with overlapping communicators leading to a deadlock.

5.2 Concept

In this chapter we present simple ideas for a layer between the application and MPI to be able to support simulated process faults. In case of no present simulated process faults, we want MPI to behave as if there was no layer between the application and MPI. To distinct between MPI functions and the corresponding functions in the simulated fault layer, we name the layer functions with the prefix SIM_. In this section we first take a look at the basic ideas and in the next section we describe the actual implementation (more detailed).

5.2.1 Blocking send/recv

Blocking functions are most straightforward. In case we have no simulated process fault, the design of MPI requires that an MPI_SEND has a matching MPI_RECV on another node in the communicator (as well as any MPI_RECV has a matching MPI_SEND) - assuming the application itself is written correctly.
For a first simple approach we don't take into account that MPI_SEND and MPI_RECV might interact with other MPI functions like MPI_PROBE.

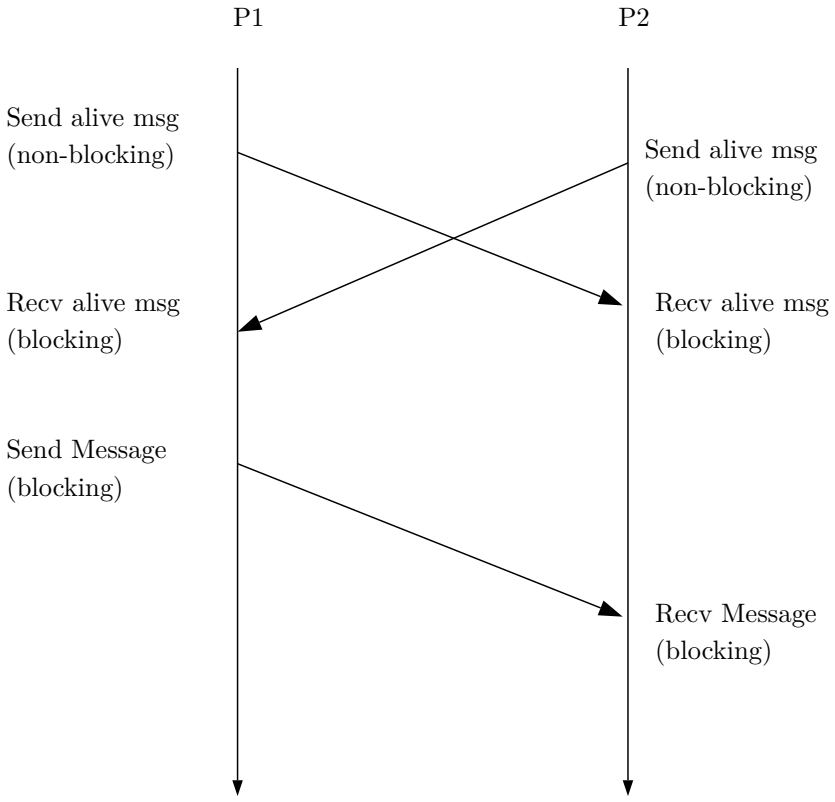


Figure 5.3: Blocking send to an alive process (Case 1).

Case 1: A process initiates a SIM_MPI_SEND to an alive process in the communicator

(See Figure 5.3) After initiating SIM_MPI_SEND, process 1 sends an alive-message to process 2 and waits for an alive-message from process 2. Process 2 does the same the other way around after initiating the matching SIM_MPI_RECV. After both processes received the alive-message, the actual MPI_SEND and MPI_RECV are executed.

Case 2: A process initiates a SIM_MPI_SEND to a dead process in the communicator

(See Figure 5.4) Process 1 sends an alive-message to process 2 and waits for its response. Process 2 responds with a dead-message and process 1 returns the MPI call with the error MPI_ERR_PROC_FAILED.

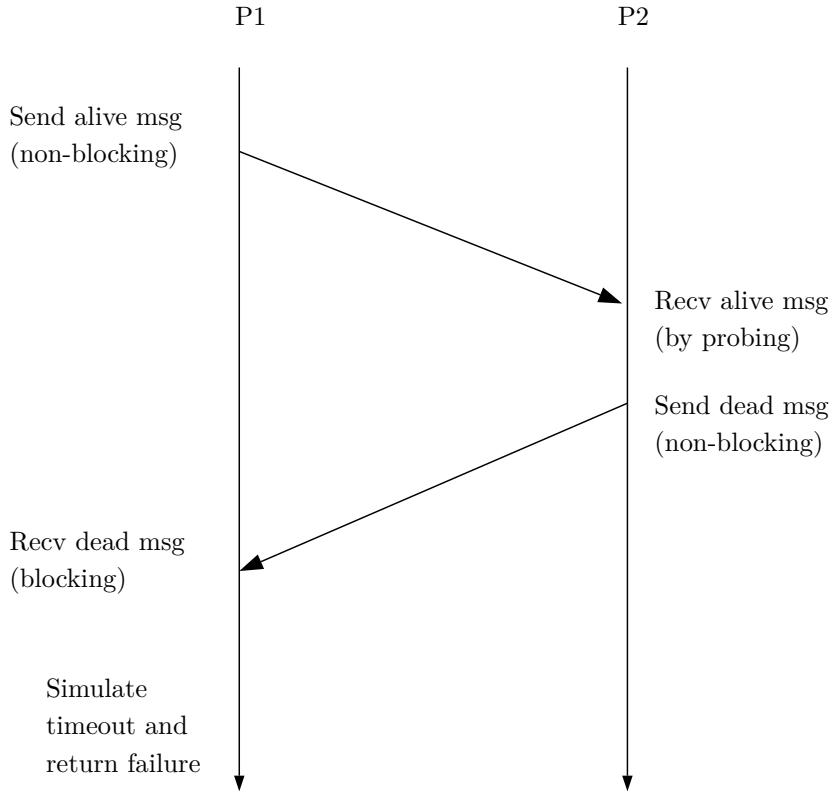


Figure 5.4: Blocking send to a dead process (Case 2).

Case 3: A process initiates a SIM_MPI_RECV to receive a message from a dead process

The procedure is the same as in Case 2.

Case 4: A process initiates a SIM_MPI_RECV to receive a message from any process (MPI_ANY_SOURCE) in a communicator with no dead processes

In this case we can't just "synchronize" with another process to know about its alive status. Instead we probe for messages. If an incoming message is detected by probe, receive it by calling MPI_RECV and its return code is returned by SIM_MPI_RECV.

Case 5: A process initiates a SIM_MPI_RECV to receive a message from any process (MPI_ANY_SOURCE) in a communicator with one or more dead processes

The procedure is the same as in Case 4, but instead of receiving the message, we get notified of a present dead process in the communicator, stop probing for incoming messages and return the error MPI_ERR_PROC_FAILED.

5.2.2 Blocking collective functions

As with the blocking send/recv we are guaranteed, that a blocking collective operation is called at every (alive) node in a communicator.
We use this fact and exchange the information through the whole communicator, which processes are currently dead, right before an actual collective MPI operation should be executed. If there are one or more dead processes, the collective operation is not executed and the call is returned with the error MPI_ERR_PROC_FAILED. If all processes are alive, the intended collective MPI operation is executed and the return code of this operation is returned to the application.

For our synchronization of how many processes are dead right before the blocking operation should begin, we require the synchronization protocol to be strong consistent. The reason for this is, because we intend to execute the actual blocking MPI operation and a collective operation cannot be canceled in MPI (especially a blocking one), we require either none or all processes in the communicator to participate.

5.2.3 Non-blocking collective functions

Non-blocking collectives are sparsely more complicated. While the first idea might be to immediately begin a communication and cancel it if a participating process is dead, this is unfortunately not possible, because non-blocking collectives cannot be canceled with MPI_CANCEL.
A non-blocking MPI operation call has to return immediately, so we cannot synchronize our knowledge about present dead processes prior to the non-blocking call. We therefore immediately execute any non-blocking collective function after called by the application and check for dead processes later within a completion function.

If a process fault is detected in a completion call and the detected dead process never called the corresponding non-blocking collective function, it is assumed that the collective operation is never going to be completed (that is true especially for functions like MPI_IBARRIER - where every single rank in the communicator has to participate before the call can be completed). The MPI request is then completed by MPI_REQUEST_FREE and the completion function returns the error MPI_ERR_PROC_FAILED.

Because the ”checking” for dead processes is done in a completion operation and the collective operation is called even if one or more processes are dead in the communication, we only need our synchronization protocol to be weak consistent. That means eventually all processes know about every present dead process in the communicator.

5.2.4 Non-blocking send/recv

We have the same situation as with non-blocking collectives. Unlike the blocking version we cannot ”check”, if a process is alive or dead before a send or receive operation is started. A non-blocking send/recv has to return immediately.
Because of that, our layer also forwards a message by MPI_ISEND or initiates a receive by MPI_IRECV immediately after function call. We use the fact that a non-blocking operation has to be completed by either MPI_TEST, MPI_WAIT or MPI_REQUEST_FREE. The ”alive-checking” is done when the application calls such a MPI completion function.

Essentially the non-blocking send and receive operations use the same approach as the one from non-blocking collectives.

5.3 Implementation: functions and description

For the implementation we want the simulated-faults-layer to have the least impact on the application as possible. On the one hand a programmer should only have to include the header files 'MPI-FT.h' and 'MPI-FT_redefine.h' instead of 'MPI.h' to enable the fault simulator. On the other hand (as already mentioned) if no process is killed by calling Sim_FT_kill_me and no ULFM functions are used, the application shall behave as if no fault simulator was active.

Redefine MPI functions
In the layer we have customized versions of MPI functions using MPI communicators or MPI requests. It is unreasonable for a programmer having to replace all MPI functions in an application with the layer versions. For that reason by including 'MPI-FT_redefine.h', all important MPI function names are replaced by using the preprocessor directive '#define' (e.g. '#define MPI_Send Sim_FT_MPI_Send'). The programmer doesn't have to worry about anything and continues using the MPI functions as if there was no layer.

Use custom versions of MPI_Comm and MPI_Request
In our fault layer, for a MPI_Comm object we need to have access to much more information than "only" the MPI_Comm object. In our header 'MPI-FT.h', we define a C++ struct Sim_FT_Comm_struct containing variables like the real MPI_Comm object used by the application, some copies of the MPI_Comm object (created with MPI_COMM_DUP) used by our fault layer, some variables needed for a custom tree structure (also see section 5.3.5.1), a list of all currently detected dead processes in the communicator and much more.
We want our custom MPI communicator to behave like the real MPI_Comm, that means for example it should not contain information directly. Instead it should behave like a pointer (similar to the real MPI_Comm) and all information is stored at the pointed location. Our custom MPI_Comm object (Sim_FT_MPI_Comm) is therefore simply a pointer to a Sim_FT_Comm_struct, which replaces the native MPI_Comm using a simple preprocessor define.

```
typedef Sim_FT_Comm_struct* Sim_FT_Comm;
#define MPI_Comm Sim_FT_Comm
```

If now the user declares a new MPI communicator object in the application, instead a pointer to our custom communicator struct is declared. On creation of a new communicator, the struct object is dynamically created using C++ new and the custom MPI communicator points to this newly created object. If a communicator is deleted by MPI_COMM_FREE, the dynamically created struct is deleted using C++ delete and the pointer is set to nullptr (which is in our layer equal to MPI_COMM_NULL: if the user compares a nullptr custom comm object with MPI_COMM_NULL using ==, it will return true).

Store currently active communicators
At some points in our implementation we need to be able to get the informa-

tion, in which communicators a specific process is currently active. For that we have a global available std::set object containing pointers to all custom communicator objects the local process is active. Every time a new communicator is created, for example by calling MPI_COMM_SPLIT, the newly created communicator 'newcomm' is added to this set. If a communicator is destroyed by calling MPI_COMM_FREE, the reference to the communicator object is deleted from the active list. This is required for example if a process calls Sim_FT_kill_me and has to iterate through all active communicators.

5.3.1 Blocking send/recv

While it is true that MPI guarantees FIFO (first in first out) delivery of messages between two processes, this only applies if the communicator and the tag are the same (1). Additionally it is possible to receive a message sent by the non-blocking MPI_ISEND with the blocking MPI_RECV (and the other way around with MPI_IRECV and MPI_SEND) (2). For the implementation we have to take into account that one process might have multiple outstanding incoming messages simultaneously, especially with different tags (3).

We don't want alive-messages to interfere with normal MPI communication from the application. Combining the facts (1) to (3), for sending and receiving the alive-message, we cannot use the same tag for all alive-messages because with multiple outstanding messages to receive we wouldn't know which alive-message belongs to which MPI message, possibly leading to a deadlock (see Figure 5.5 for a simple example). A solution for this problem is to duplicate the communicator c_comm to c_comm_copy_p2p and use the copy merely to send alive-messages. This way point-to-point alive-messages are delivered in the same order as their corresponding "real" MPI messages (note this problem only occurs because MPI_RECV can receive from MPI_ISEND; if blocked and non-blocked messages were separated, this would not be necessary).

5.3.2 Blocking collective functions

For the implementation we use an operation similar to MPI_IALLREDUCE to synchronize dead processes across the communicator prior to any "real" MPI collective. The call 'Sim_FT_Check_dead_processes' consists of a reduce, combined with a broadcast. We have two possible cases (note however there is an additional case if the communicator is revoked, this will be discussed in section 5.3.8.3).

Case 1: No process in the communicator is dead

The simulated fault layer for blocking collectives consists of two steps.
Step 1: All ranks in the communicator initiate a reduce operation with the message '0' to the root rank of the communicator.

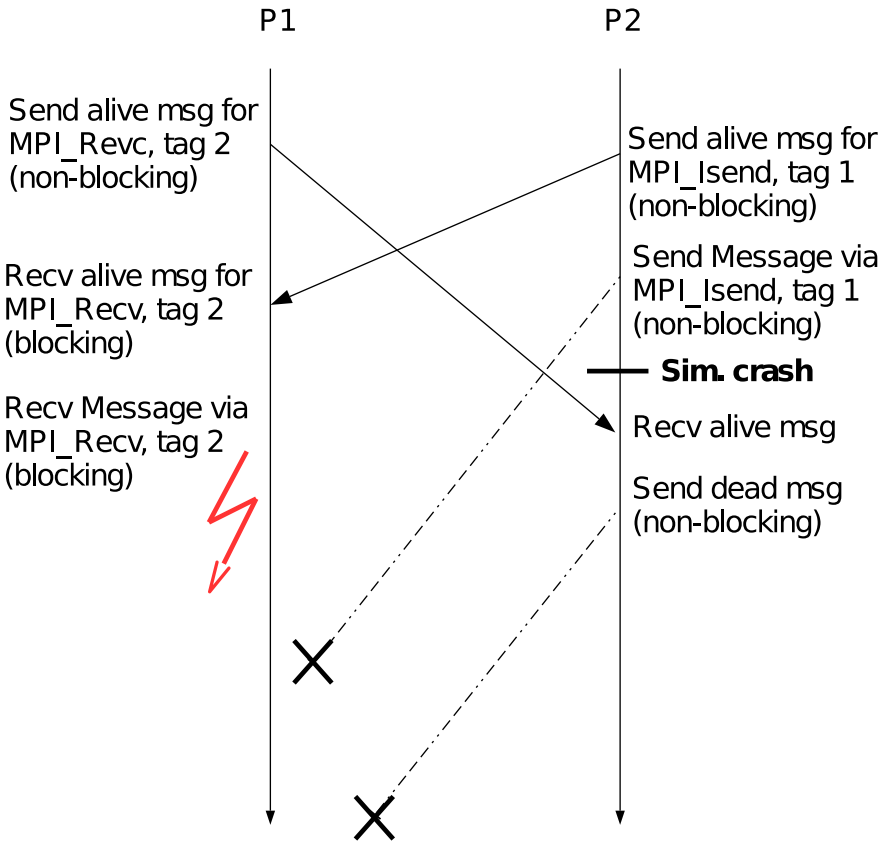
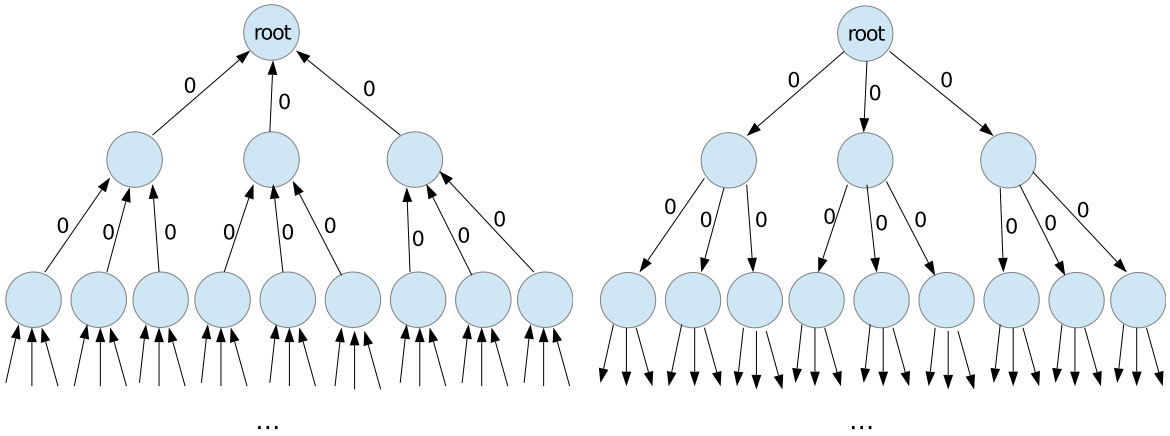


Figure 5.5: Blocking and non-blocking point-to-point can be mixed in MPI. Possible deadlock if alive-messages are transfered without tag. Example where P1 receives an alive-message belonging to an MPI_ISEND on another tag and starts receiving a non-existent message.

Step 2: Root rank broadcasts the information 'no processes are dead' to all other ranks in the communicator.
Step 3: The collective MPI function is executed at all ranks in the communicator.

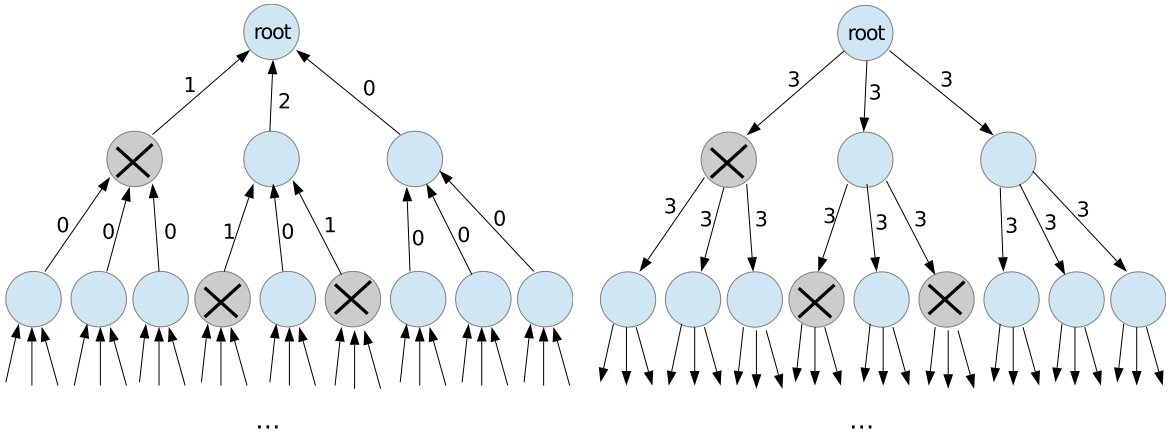
Case 2: One or more processes in the communicator are dead

Step 0: After a process calls Sim_FT_kill_me, it immediately sends a notification of its dead status to the root process of every communicator the dead process is currently active.
Step 1: All ranks in the communicator reduce the count of current dead processes in the communicator to the root rank of the communicator.
Step 2: Root waits for incoming dead messages until the messages from all dead processes are received.
Step 3: Root rank broadcasts the count of dead processes and a list including the ranks of all dead processes to all members of the communicator.
Step 4: Every alive process returns the MPI call with error MPI_ERR_PROC_FAILED and adds the dead processes to a local list of dead processes.



Step 1: Reduce Step 2: Broadcast

Figure 5.6: Collective operation without dead processes (Case 1).



Step 1: Reduce Step 2: Broadcast

Figure 5.7: Collective operation with 3 dead processes (Case 2).

5.3.3 Non-blocking send/recv

As mentioned above, any layer functions with the intention to check for present dead processes regarding non-blocking MPI operations can only be weak consistent to preserve the non-blocking behavior of these MPI operations. We have two possible cases. For readability we assume for Case 1 and Case 2 the local called operation is MPI_ISEND. For MPI_IRECV the procedure is similar with switched roles of MPI_ISEND and MPI_IRECV. The special Case 3 is specifically for MPI_IRECV.

Case 1: The point-to-point partner node is alive

Step 1: After calling SIM_FT_MPI_ISEND, an alive-message is sent non-blocking to dest using the tag of the message and using communicator c_comm_copy_p2p. Immediately after sending, the actual non-blocking message operation MPI_ISEND is called using the parameters of SIM_FT_MPI_ISEND and communicator c_comm.
Step 2: dest calls SIM_FT_MPI_IRECV, sends an alive-message and calls MPI_IRECV (similar to Step 1).
Step 3: After calling a completion function, the alive-message of dest is received and the call is completed.

Case 2: The point-to-point partner node is dead

Step 0: After the dead process calls Sim_FT_kill_me, a dead-message is sent to the root of every communicator. Root initiates a broadcast containing a list of all recently detected dead processes, including dest.
Step 1: (same as Step 1 of Case 1).
Step 2: The dead process (dest) receives the alive-message and ignores the message. The alive process receives the broadcast from root and adds the partner node to a local list of dead processes. Any upcoming completion operation (MPI_TEST, MPI_WAIT,...) concerning a message to or from the partner node is returned with the error MPI_ERR_PROC_FAILED.
Step 3: The alive process sends an "command message" to the dead one, ordering the dead process to receive the outstanding MPI_ISEND by calling a matching MPI_IRECV. The purpose of this message is to complete the actual sent message and free resources.

Case 3: MPI_IRECV with source MPI_ANY_SOURCE and a present dead process in the communicator

In this case, after calling a completion operation, the function is returned with the error MPI_ERR_PROC_FAILED_PENDING.

5.3.4 Non-blocking collective functions

Collective operations are guaranteed to be executed in the same order at all ranks in the communicator [MPI15, Example 5.30]. That is for example in a communicator with two processes (rank 0 and rank 1) if rank 0 first calls an MPI_IBARRIER, then an MPI_BARRIER and finally an MPI_IBCAST, rank 1 has to execute these functions in the exact same order.
By design of our fault simulator, we are guaranteed that a blocking collective operation is only executed if no processes are dead. It is therefore not possible to have a blocking collective active only on a subset of a communicator.

In our customized communicator object, we store for each process locally its recently executed non-blocking collective operations since the last blocking collective

was executed.
We use a list containing objects with the information, which non-blocking collective was executed and what its parameters were, for example the operations' root rank, how big the collective message was (e.g. in case of MPI_IBCAST) or which operation was used (for MPI_IALLREDUCE or MPI_IREDUCE). This object might be extended to support tracking more non-blocking collective operations.
The purpose of this list is to be able to complete outstanding non-blocking collectives on the one hand and to free blocked processes waiting for the completion of a non-blocking collective that is never going to be completed because a dead process never executed the corresponding non-blocking collective operation on the other hand. The actual messages of these operations are not stored, because in order to free an operation it is sufficient to send a dummy message with arbitrary values - only the size of the message is important.

Let's take a look at how non-blocking collective functions are executed in the fault simulator, beginning with the call of the function and ending with the completion of the collective operation. For readability we call the non-blocking collective function MPI_ICOLLECTIVE.

We have three possible cases regarding the execution of non-blocking collective functions, all beginning with the same Step 0:

Step 0: After calling SIM_FT_MPI_ICOLLECTIVE, the corresponding collective MPI_ICOLLECTIVE is executed immediately and its parameters are added to the local list of recent non-blocking collectives (as mentioned at the beginning of this section). The return value of MPI_ICOLLECTIVE is then returned by SIM_FT_MPI_ICOLLECTIVE. Note that no checking for failed processes is done here in order to be consistent with the proposed ULFM standard.

In the following let's assume the collective operation is to be completed with MPI_TEST. For simplicity we say 'NBC' short for 'non-blocking collective'.

Case 1: No process in the communicator is dead at the time of completion

Step 1: After the completion function SIM_FT_MPI_TEST is called, the fault simulator calls MPI_TEST and returns its return values. If the operation was completed, the custom request object is deleted using C++ delete and the request pointer is set to nullptr.

Case 2: One or more processes in the communicator are dead at the time of completion and the NBC count of at least one dead process is lower than the local NBC count

Step 1: Immediately after a process goes "dead" by calling Sim_FT_kill_me, it sends a message to the root rank of each communicator containing the current size of its list of recent non-blocking collective operations.
Step 2: Root broadcasts this count to all ranks in the communicator if no count has been broadcasted yet or the count is lower than the count from the last broadcast.
Step 3: The current process receives the broadcast and compares the NBC count from the broadcast with the local NBC count. Because the local count is higher than the one from the dead process, it assumes the dead process never participated in the non-blocking collective and thus the operation will not be completed. The completion function is returned with the error MPI_ERR_PROC_FAILED.
Step 4: After calling MPI_COMM_FREE on the current communicator, all outstanding non-blocking collectives are completed at all processes in the communicator, including the dead ones (also see section 5.3.10 for more details about how the collective operations are completed).

Case 3: The communicator is revoked

Step 1: If the communicator is revoked, we have to assume that the non-blocking collective is never executed on one or more other processes in the communicator. As specified in the ULFM standard, the SIM_FT_MPI_ICOLLECTIVE function call is then returned with the error MPI_ERR_COMM_REVOKED and the collective operation is not executed.

5.3.5 Background broadcast

For our layer it is important to be able to send messages in the background through the whole communicator. One way to do this would be directly sending a non-blocking point-to-point message to all other ranks in the communicator. Whereas for a small communicator this approach would be totally fine, bigger communicators would lead to performance issues because this basic approach doesn't scale.
As mentioned in chapter 1.3, we use two different scaling approaches to send messages in the background. One using non-blocking collective functions and the other using a custom implementation of non-blocking collectives only consisting of non-blocking point-to-point operations.

5.3.5.1 Approach without non-blocking collectives

To be able to run our fault simulation layer with MPI implementations not (fully) supporting non-blocking collectives, we use a custom version of non-blocking MPI _IREDUCE and MPI_IBCAST only consisting of non-blocking point-to-point messages.

When a communicator is first initialized, for example after calling MPI_COMM_SPLIT, we have to initialize our custom communicator in the layer by calling Sim_FT_Initialize_new_comm. On initialization we create a very simple tree topology (described below) and saving locally for every process in the communicator object, which ranks its successors are and which rank its predecessor is. The tree uses the root as specified in the communicator object (default rank 0 in the communicator).

Having a tree topology, we can use this topology to create our custom non-blocking collectives. Note that dead processes use slightly different versions of these collective operations than alive processes, because dead processes initiate the operations in advance and alive processes only initiate operations when a collective MPI operation is to be executed.

Tree topology

This very basic approach of creating a tree topology does not take into account the topology of the underlying network. Assume root of the tree has id 0 and every node in the tree shall have two (or less) children. Then the successors of root would be rank 1 and 2. Rank 1 would have the successors 3 and 4, while rank 2 would have the successors rank 5 and 6, etc.

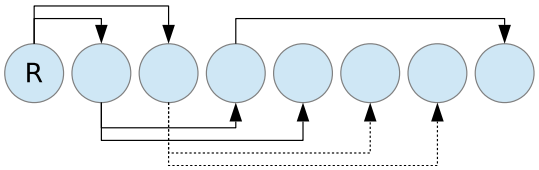


Figure 5.8: Simple tree topology with 8 nodes and max. 2 child nodes.

The calculation of the predecessor and the successors is very simple. The calculation of the successors is described in Algorithm 1.

Algorithm 1 Calculate successors

```
1:  procedure BCAST_GET_SUCCESSORS(root, id, tree_size, Successor_Count)
2:      for i ← 0, Successor_Count − 1 do
3:          k ←NORMALIZEID(id,root)      ▷ Shift process ids s.th. root has rank 0.
4:          k ←CONVERTID(k, tree_size)              ▷ Make sure 0 ≤ k < tree_size
5:          j ← k∗Successor_Count + i + 1
6:          j ←DENORMALIZEID(j)          ▷ Revert the shift from the beginning
7:          j ←CONVERTID(j, tree_size)              ▷ Make sure 0 ≤ j < tree_size
8:          Add j to the vector of successors from id
9:      end for
10: end procedure
```

In the algorithm, tree_size denotes the number of ranks in the communicator. The assignment is unique, so the inversion (calculation of the predecessor) is just as

simple: assuming the id is already normalized, the predecessor is calculated by the following formula

$$\text{PredecessorID} = \frac{\text{id} - 1 - (\text{id} - 1) \bmod \text{Successor\_Count}}{\text{Successor\_Count}}$$

which has to be denormalized afterwards.

These functions are called every time a new communicator is created. The successors and predecessors for every process in the communicator are then permanently stored in the custom communicator object.

Custom IBCAST

The custom non-blocking broadcast can send messages via specific tags using a special copy of the regular communicator to not interfere with MPI communication from the application. Every process in the communicator has to frequently call the function 'Sim_FT_Perform_background_operations' to forward incoming broadcast messages. The custom broadcast waits for a non-blocking point-to-point-message from the current process' predecessor. After receiving the message, it forwards the message to all successors ("child nodes") in the tree by sending a non-blocking point-to-point-message using the same tag.

Custom IREDUCE

The custom ireduce is a little bit more complicated because we cannot receive a message and forward it in the same step. We need to save the information, how many successors have already sent a message and perform a reduce operation for every received message. After all successor messages are received, the reduced message is then sent to the predecessor via a non-blocking point-to-point-message.

There are two slightly different versions of the custom non-blocking reduce. One version is used by alive processes, it behaves like a blocking reduce with one important difference, that while waiting for incoming messages, our layer function Sim_FT_perform_background_operations is called continuously.
The other version is used by dead processes. This version has to be non-blocking, because a dead process iterates through many different communicators and has to maintain multiple custom non-blocking reduce operations simultaneously. For this reason, the current reduce message and the information, which successor message is already received, is stored inside the custom communicator object. If a new successor message is received, the stored message gets "merged" together with the received message by a reduce operation.

5.3.5.2 Approach using non-blocking collectives

Note that due to compatibility issues, this approach is not implemented. Nevertheless we take a look at the basic idea because of the possible performance boost.

While the custom implementation of a non-blocking broadcast is function-wise totally fine, the tree created does not take the topology of the network into account. A successor of the custom topology could therefore be far away in the network, leading to high delay and low transfer speed, while there could be multiple nodes locally on the same machine with almost zero delay and instant data transfer.
Additionally a message in the custom broadcast is only forwarded if the successor received the message and calls the custom broadcast function, where a MPI-internal non-blocking broadcast could be optimized to forward a message through the communicator even if a broadcast function is not called on all nodes in the communicator.

Now returning to the MPI non-blocking collective functions. The idea is very simple: in our custom communicator object, we have defined a root rank and a MPI_REQUEST object for every background broadcast we intend to use. On the initialization of a communicator object, each rank in the communicator except root begins a MPI_IBCAST. For example if the background message is to be 'comm is revoked' and the used communicator object is 'comm', the call looks like

```
MPI_Ibcast(0, 0, MPI_INT, comm->Root_Rank,
                comm->c_comm_copy_collective, comm->Revoke_Request);
```

If now the communicator is to be revoked, root also calls this function and after calling a completion function, the non-blocking broadcast will be executed at all ranks. If a process wants to know if the communicator is revoked, it only has to check with MPI_TEST, if the broadcast is completed (meaning, the buffer can be used again, which implicates the broadcast message is received and forwarded).
In this case the broadcast will only be executed at most once for every communicator, because a revoked communicator stays revoked until a shrink operation is called - creating a new communicator. If however the background broadcast should be able to occur again, each rank in the communicator except root calls MPI_IBCAST again immediately after the last broadcast operation is completed.

5.3.6 Kill me

If a process is to be simulated dead, it calls the function Sim_FT_Kill_me and from this point the process does not participate in the normal flow of the application anymore. Instead the layer waits for incoming alive-messages and responds them with 'I_AM_DEAD', or waits for new collective operations and participates in the reduce operation with a dead count '1' and forwards dead broadcasts. Additionally the dead process also forwards other background messages like revoke messages.
Once a process is dead, it will stay dead until the MPI network is terminated by MPI_FINALIZE.
See Figure 5.9 for a brief overview of the execution flow of Sim_FT_kill_me.

The function Sim_FT_Kill_me can either be called manually in the application or by defining specific conditions in the function Sim_FT_decide_kill, which is called prior to every implemented MPI function. A process can be killed completely random using a pseudo random number generator or after a specific amount of MPI calls.

It is also possible to kill a process randomly with growing probability, depending on when the function Sim_FT_decide_kill was called last (see for example equation 5.1 on page 52 and the related example). If the last check was some seconds ago, the probability of a recent failure is much lower than if the last check was an hour ago. Another possibility would be on initialization of the communicator to calculate the life span of a specific process using an appropriate probability distribution, and kill it if the execution time is higher than the calculated life span.

5.3.7 MPI_FINALIZE

The procedure is the same as with the ULFM function MPI_COMM_SHRINK, please see section 5.3.8.1 on page 39.

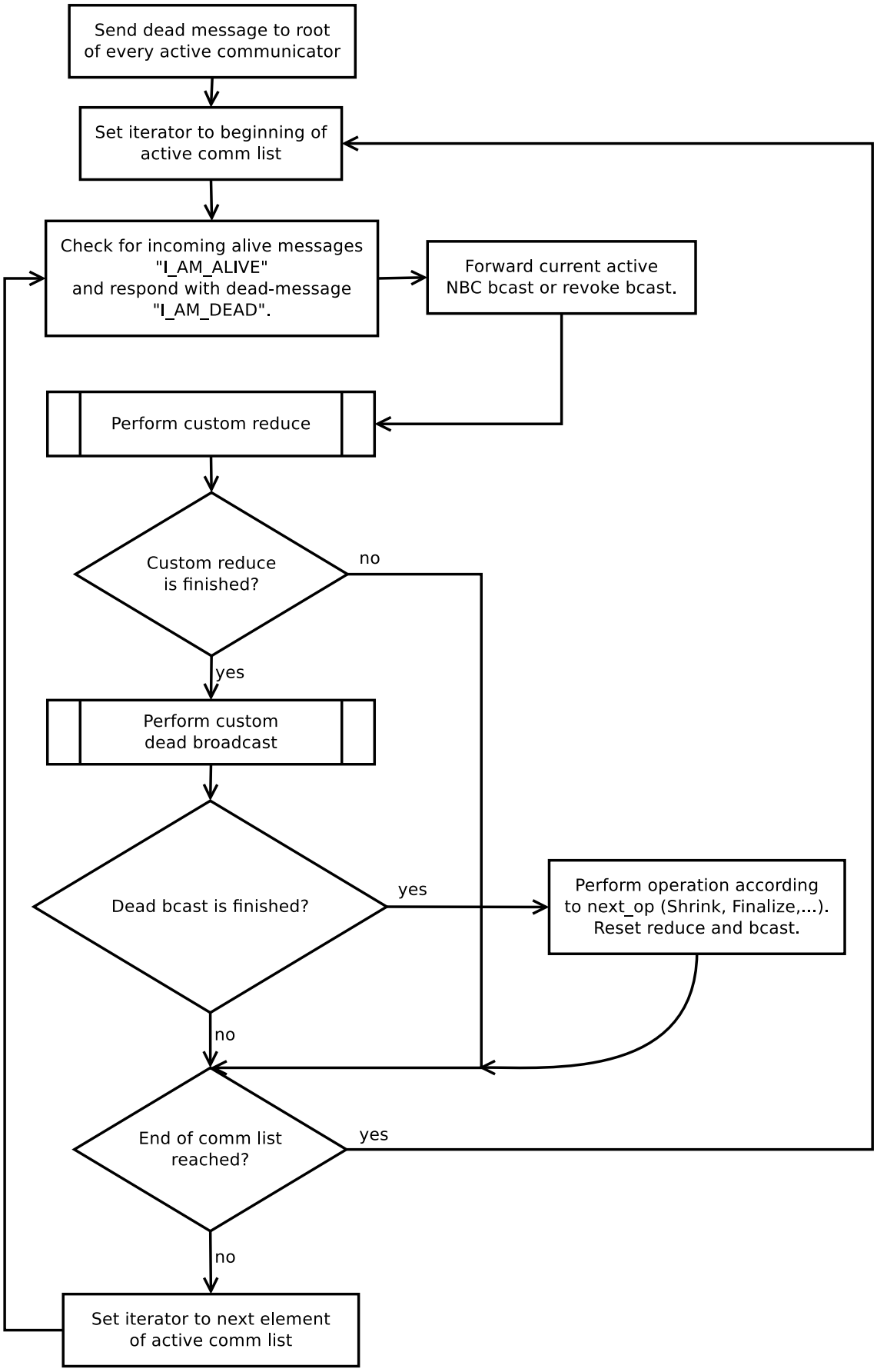


Figure 5.9: Execution flow of Sim_FT_kill_me.

5.3.8 ULFM functions

All functions discussed and implemented up to this point are merely used for supporting simulated process faults as well as propagating (and "detecting") failed processes. This section is about the implementation of a ULFM interface to give the application the ability to treat detected simulated failed processes.

5.3.8.1 MPI_COMM_SHRINK

MPI_COMM_SHRINK is a blocking collective function and thus has to be called by all alive processes in a communicator. We already have a method to distribute the current dead processes across the communicator (see blocking collectives section 5.3.2). For MPI_COMM_SHRINK we expand the functionality of our function Check_dead_processes to not only distribute the dead processes but also the information, if the following operation shall be a regular blocking MPI function (the distributed value in this case is 'Default'), a MPI_COMM_SHRINK (by distributing the value 'Shrink') or a MPI_FINALIZE (by the value 'Finalize'). We need this information available at all ranks because these special collective functions require the participation of all processes - including the dead ones.
While this function only makes sense if at least one process in the communicator is dead (Case 2), it is totally fine to call it if all processes are alive (Case 1). The procedure in our implementation is also the same, but for a better illustration we distinguish these two cases.

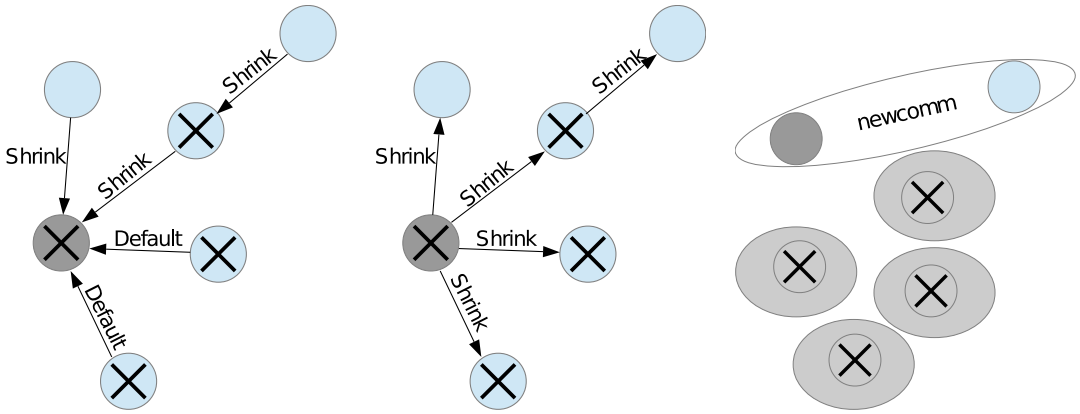
Case 1: No process in the communicator is dead

Step 1: After calling MPI_COMM_SHRINK, all processes in the communicator initialize a custom reduce operation with the information 'Shrink'.
Step 2: Root broadcasts the information 'Shrink' to all processes in the communicator.
Step 3: All processes call MPI_COMM_SPLIT with the color '0' and their current rank as key, creating a new communicator containing all processes of the old communicator.

Case 2: One or more processes in the communicator are dead

Step 1: After calling MPI_COMM_SHRINK, alive processes initialize a custom reduce operation, attaching the information 'Shrink'. Dead processes forward this information to their predecessor in the tree. This step is especially important if root itself is dead. Then the only way for root to know about an upcoming Shrink is by receiving this reduce.
Step 2: Root broadcasts the information 'Shrink' to all processes in the communicator. The broadcast guarantees that dead processes at the leafs of the tree also receive the information 'Shrink'.
Step 3: After receiving and forwarding the broadcast message, all processes call

MPI_COMM_SPLIT. Alive processes use the color '0' and their current rank as key, dead processes use the color 'MPI_UNDEFINED' and their current rank as key. The newly created communicator contains only alive processes, dead processes are excluded.



Step 1: Reduce

Step 2: Broadcast

Step 3: Split. newcomm contains only alive processes.

Figure 5.10: (Case 2) Shrink with 4 dead processes (including root) and 2 alive ones.

5.3.8.2 MPI_COMM_AGREE

We use the same mechanic as in MPI_COMM_SHRINK (section 5.3.8.1). We have two special values 'AFalse' and 'ATrue'. In the reduce phase of our Check_dead_processes, these values are combined by a bit wise 'AND' operation. In the broadcast phase, the same "agreed" value is broadcasted to the communicator and returned by all alive functions.

5.3.8.3 MPI_COMM_REVOKE

After calling MPI_COMM_REVOKE on a communicator, a non-blocking revoke-message is directly sent to the root node of the communicator, if the communicator is not revoked yet. Root receives the revoke message (this receive is part of the function perform_background_operations) and initiates a custom background revoke-broadcast to the communicator. Once a process receives the broadcast, it sets the variable 'Revoked' in the custom communicator to true and afterwards returns all point-to-point and collective operations with the error MPI_ERR_REVOKED.

As an addition to the discussed cases in section 5.3.2, after a comm is revoked, Check_dead_processes uses different tags for its custom reduce and custom broadcast. This is done so that it is not possible for some processes to initiate a dead sync operation, while others have already received their revoke messages and never participate in those operations, leading to a deadlock.
Already started reduce operations with the "old" tag are then directly responded

with a broadcast, informing the sender about the revoked communicator. This is especially important for reduce messages initiated by dead processes. By accepting the messages and responding with 'the communicator is revoked', the dead processes can switch the used tags to the new ones.

Note that this implementation does not scale and could flood the root node in huge communicators, if multiple processes revoke the communicator at the same time. Also currently the call might lead to a dead lock, if not treated right. If in our fault simulator one process calls a blocking collective MPI function, all other alive processes are required to call the blocking collective function as well in order to "unblock" the processes that already called the collective function. If a revoke message is received, the programmer has to know, which blocking collective is called next in the application and call it at all ranks.

5.3.9 MPI_PROBE

Additionally MPI_PROBE was implemented, but only to receive messages from blocking MPI_SEND. Because messages are not sent directly in our layer, the layer first sends a probe request to the sending process via the same channel as alive-messages. The sending process receives the request and responds over a special probe communicator with a message containing the size of the message to be sent. Then, the probing process receives the count and returns the call. MPI_GET_COUNT returns then the count received from this message.

5.3.10 MPI_COMM_FREE and free outstanding non-blocking operations

MPI_COMM_FREE is in our fault simulator required to be called as if it was a blocking collective operation, that means it has to be called by all alive processes simultaneously. We use MPI_COMM_FREE not only to free (delete) the communicator, but also to complete all outstanding non-blocking collective operations. Some operations might have been started at multiple processes, while a possible dead process never participated. According to the MPI standard, all non-blocking collective operations have to be completed.

A simple test (create a new comm with MPI_COMM_SPLIT, initiate the non-blocking MPI_IBARRIER at one process in the comm and try to free the comm at all ranks) using MPICH MPI revealed, that by calling MPI_COMM_FREE, outstanding operations are indeed NOT completed. Additionally the communicator object does not get freed. By executing the test in a loop, the application terminates after some time with an error stating that the maximum amount of communicators is reached (in our case 2048 communicators) and no more can be created.

Behavior in our fault layer

The following steps describe the behavior of our SIM_MPI_COMM_FREE.

Step 1: A simple synchronization operation (check_dead_processes) is executed with next_op Default. We do this because after this operation the process with the highest count of recent NBC [1] operations is known to root.

Step 2: Root sends a request message to the process with the highest count of recent NBC operations.

Step 3: The process responds the request with a serialized version of its local NBC vector.

Step 4: Another synchronization operation is startet, but this time with next_op 'Commfree'. Root attaches the received serialized NBC vector to the synchronization broadcast and after this operation, all processes (including the dead ones) have the same NBC vector available. This "remote" vector has an equal or greater size than the local recent non-blocking collectives vector. In MPI all NBC operations have to be executed in the same order at all ranks in the communicator, that's why the first entries of the remote vector are identical to the entries of the local vector. (also see Figure 5.11)

Step 5: Let n be the size of the local recent NBC vector. Each process in the communicator begins executing the NBC operations stored in the remote NBC vector, beginning with the entry $n + 1$.

Step 6: Each process completes all operations. For the first n operations, the request object from the local NBC vector is used (because they were initiated prior to the completion protocol). For the rest, the request object from the remote NBC vector is used (which were initiated in Step 5). The custom request objects are then deleted using C++ delete (because they were created dynamically using C++ new).

Step 7: The MPI communicator and all its copies created by our fault layer are deleted using MPI_COMM_FREE. The custom communicator is deleted from the local list of currently active communicators and the custom communicator object is deleted using C++ delete.

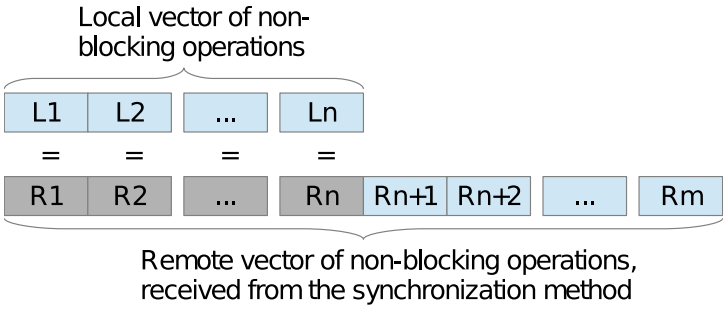


Figure 5.11: Local and remote vectors containing the information, which non-blocking operations were started, including the necessary parameters to complete them.

[1] non-blocking collective

5.3.10.1 Detailed example

To get a better understanding of the functioning, we take a look at a detailed example. In our example we have four processes in the same MPI network and the same communicator (other than MPI_COMM_WORLD, because otherwise we couldn't free the communicator). Each process executes the same MPI program (see Algorithm 2 for an excerpt of our very basic fictitious program).

Algorithm 2

```
1: MPI_Ibarrier
2: MPI_Wait                    ▷ Wait for Ibarrier to complete
3: MPI_Ibcast                  ▷ With root process 1 and any message
4: MPI_Wait                    ▷ Wait for Ibcast to complete
5: MPI_Ibarrier
6: MPI_Wait                    ▷ Wait for Ibarrier to complete
7: MPI_Comm_free               ▷ Free the comm where the operations were called
```

Right before calling MPI_Ibcast (line 3), process 2 crashes. The following 10 Figures illustrate the behavior of the processes step-by-step. The number next to the processes indicates the currently executed line of the fictional algorithm. Process 0 is the root process of our custom communicator.

Behavior of our processes step-by-step

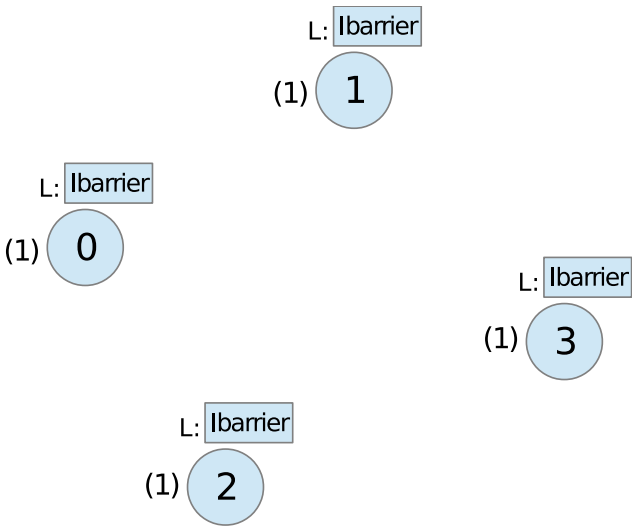


Figure 5.12: (1.) All processes call MPI_Ibarrier. The call is stored in the local list of recent non-blocking collectives.

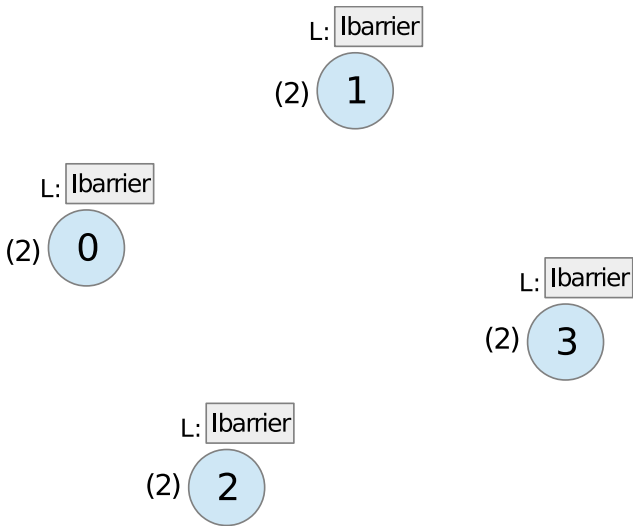


Figure 5.13: (2.) There are currently no failed processes, so after calling MPI_Wait the Ibarrier is successfully completed at all ranks.

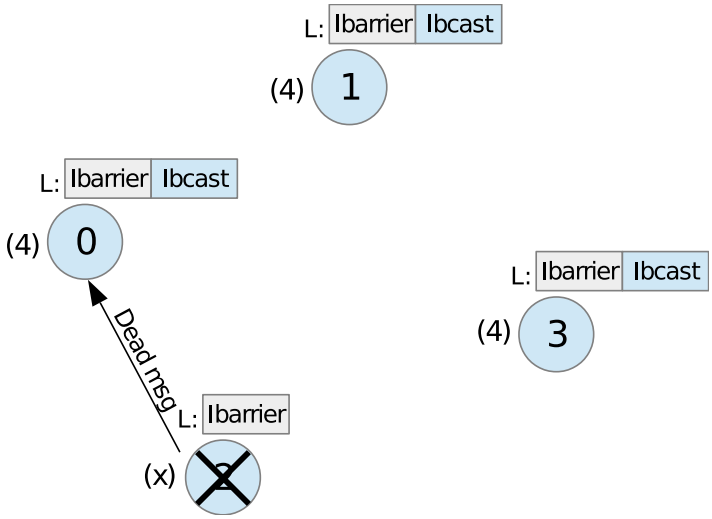


Figure 5.14: (3.) Process 2 fails and the layer immediately sends a dead message to root. The other processes have initialized an MPI_Ibcast and wait for its completion.

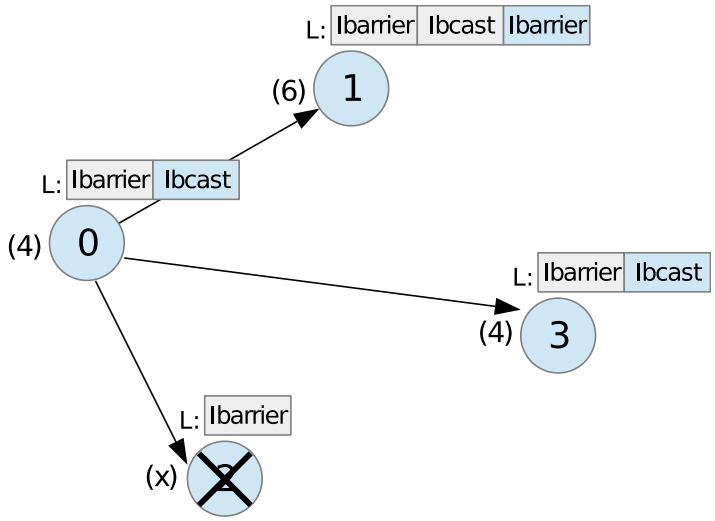


Figure 5.15: (4.) Despite the present (but undetected) dead process, MPI_Ibcast is quickly completed at process 1, because as root of the broadcast, the broadcast message is sent immediately by MPI and the buffer can be reused. After the completion of Ibcast, the Ibarrier is initialized and process 1 waits for its completion.
Meanwhile, after process 0 receives the dead message, it initializes a background dead broadbast to inform the other processes of the dead process.

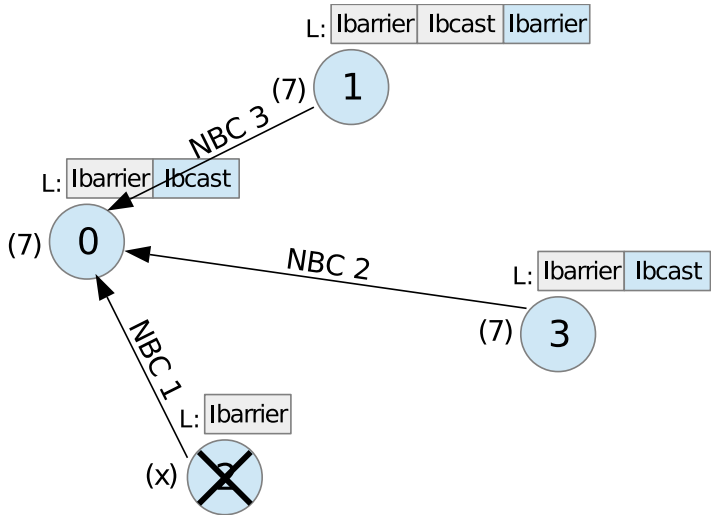


Figure 5.16: (5.) All processes received the dead broadcast and return from their MPI_Wait functions with an error. New non-blocking calls are also returned with an error.
All alive processes therefore reach MPI_Comm_free and initialize the completion protocol by reducing their current local NBC vector size.

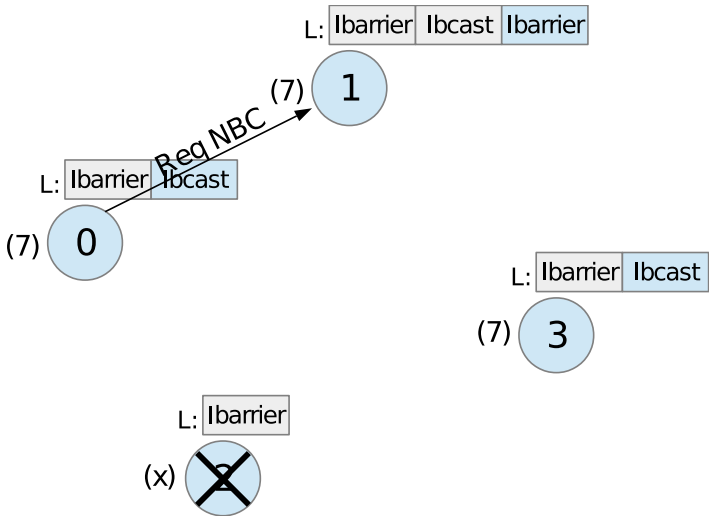


Figure 5.17: (6.) Root of the communicator requests the NBC vector from the process with the greatest vector (process 1).

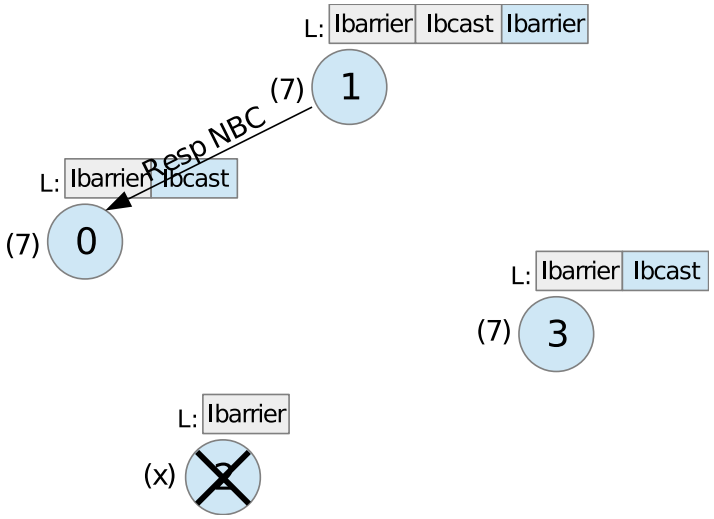


Figure 5.18: (7.) Process 1 responds the request with a message containing its local NBC vector elements.

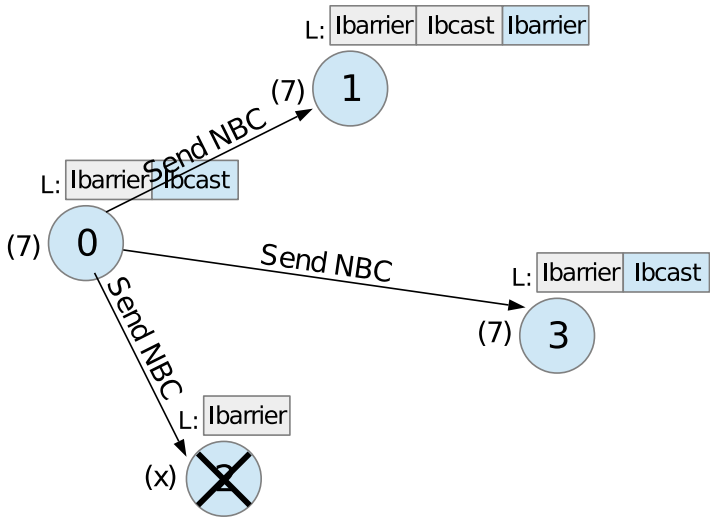


Figure 5.19: (8.) Root sends the NBC vector to all processes in the communicator.

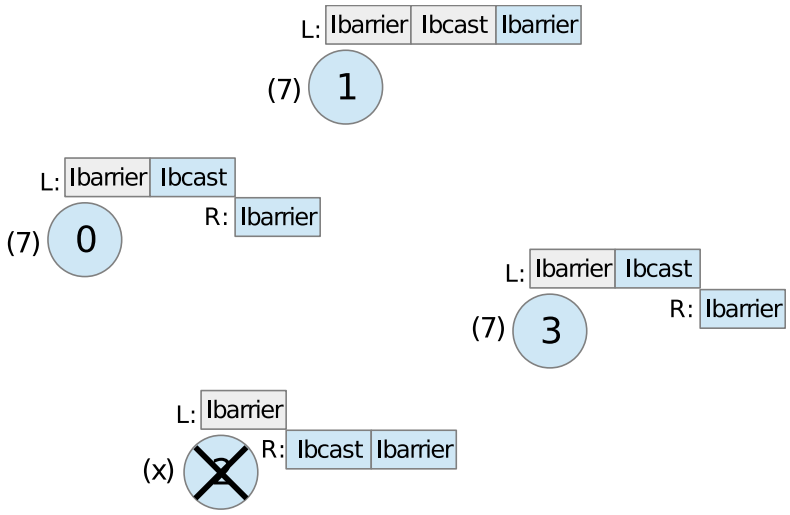


Figure 5.20: (9.) All processes receive the NBC vector and initialize non-blocking collectives newly received from the (remote) NBC vector and not yet in their local vector. All processes in the communicator have now access to the same list of non-blocking collectives.

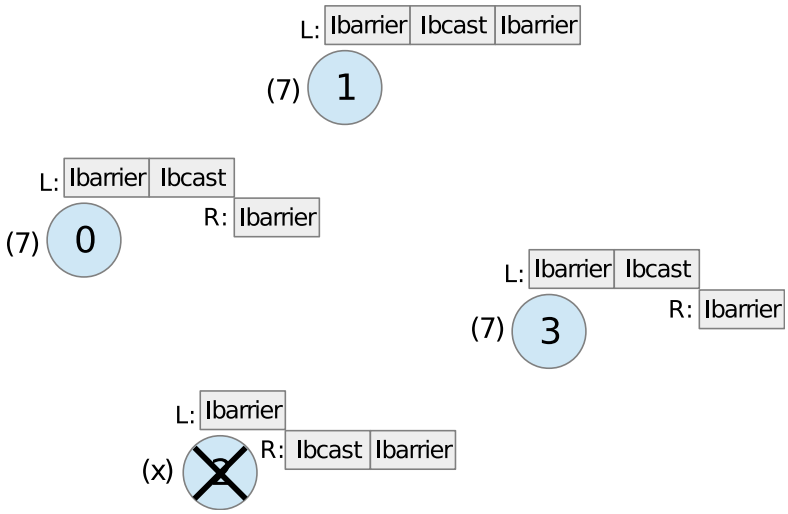


Figure 5.21: (10.) All non-blocking collectives (local and remote) are completed at all ranks in the communicator. The communicator object and all its copies are then destroyed by MPI_Comm_free. The corresponding custom communicator object is deleted from the active communicators set.

5.4 Silent faults

We implemented a fault simulator to be able to simulate process failures in an MPI network. In section 2.1 we saw other possible fault models. While faults such as performance loss due to hardware malfunction is very hard to simulate, faulty memory leading to unwanted random bit flips can be easily simulated by manipulating bits directly in MPI messages.

Possible implementation

Let's assume we have an array of n variables (like in Figure 5.22) being sent by MPI_SEND. One variable in the array has the size of k bits, so a message with a total of kn bits will be sent. In our model, given a memory module is faulty, let the chance of an arbitrary bit being flipped (that means 1 will be changed to 0, 0 will be changed to 1) be $0 < p_f \leq 1$.
If we further assume, the event 'the bit at index j is being flipped due to a fault' happens independent of any other possible flips, we can use the binomial distribution along with a uniform distributed continuous random number generator and the Inverse transform sampling to calculate, how many of our kn bits are going to flip.
After we have calculated the number of bits to be flipped (let's call the number m) we fill a std::vector with the numbers $1, 2, ..., kn$. Then we shuffle our vector randomly (for example using std::random_shuffle) and flip the bits in the message at the bit positions stored in the first m entries of the shuffled vector (or in other words: we choose m random bit positions from the set $\{1, 2, ..., kn\}$).

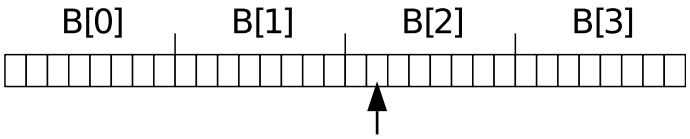


Figure 5.22: A possible representation of a char array in the memory (the values are omitted). The tagged bit has the position 17.

Our implementation Sim_FT_Manipulate_bits takes as parameters an integer array and its size, together with the probability of a bit flip. We make use of the standard C++ class binomial_distribution to generate the amount of bits to be flipped, given a specific flip probability p.

Impact of bit flips on the application stability itself

Our model is only realistic, if with growing probability p_f of bit errors, the probability of a total process failure p_p also grows. That is because bit flips could not only affect the messages, but also the system stability itself. If MPI or system critical areas of the memory are defective, the application or the operating system is very likely to crash or fail.

The probability of a process crash in a specific time range could for example consist of a constant part p_c and a variable failure part depending on p_f. Our model has some requirements on our probability values:
(1.) If $p_f = 1$, p_p should be 1. If $p_c = 1$, p_p should be 1 as well.

(2.) If $p_f = 0$, we want $p_p = p_c$. If $p_c = 0$, we want $p_p = 1 - (1 - p_f)^k$, where $k \geq 1$ is a value that should represent the size that the application uses in the memory (with greater program size, the impact of bit flips should be greater and more likely), greater k results in a higher process failure probability.

(3.) We would like p_p to be equal to or greater than p_c and equal to or greater than p_f. Also p_p should be monotonically increasing if we change the values p_f or p_c.

By combining these three requirements, our process failure probability could be

$$p_p = p_c p_f + (1 - p_c p_f) \left(\frac{p_c + p_f + (1 - p_f)p_c + (1 - p_c)(2 - 2(1 - p_f)^k - p_f)}{2} \right)$$

where p_c would be the process failure probability without taking the memory fault into account and p_f would be the probability of a single bit to be flipped. In Figure 5.23 you can see a plot of p_p using the parameter $k = 3$.

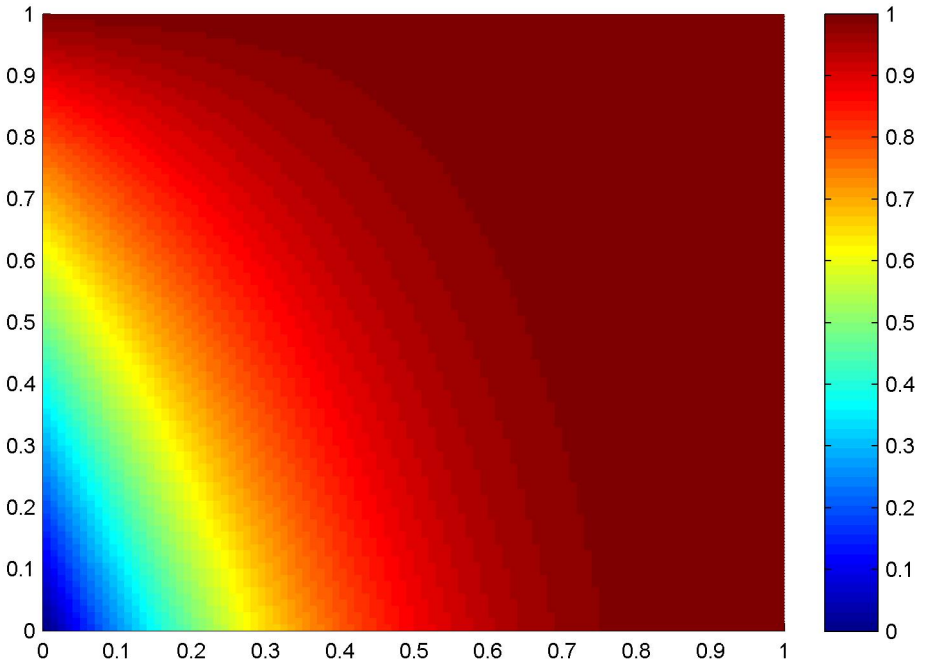


Figure 5.23: A plot of p_p created using the parameter $k = 3$. The x-axis denotes the values of p_f, the y-axis denotes the values of p_c. The color represents the process failure probability.

5.5 Other approaches

We presented a working solution to be able to simulate process faults in current MPI implementations. Our solution is a compromise between performance and compatibility. Because of the many other ways to create a fault simulator, we present possible other approaches, including clarifications why a particular other approach was not used for our implementation - this could be for example due to performance issues or missing functionality in MPI.

5.5.1 One master coordinates all failures

This is a simple approach, where a dead process reports to a coordinating master process immediately after calling Sim_FT_kill_me. If another process attempts to send a message to a dead process, it requests an alive-status from the master process and proceeds depending to the masters' answer. If the master process responds with 'the process is alive', the MPI operation is executed, otherwise an error is returned to the application. This leads to some problems concerning consistency and performance.

On the one hand a process requesting the alive-status of another process might get the wrong information 'the process is alive' due to network delay. After sending a message, the receiving process might be already dead an never receives the message, creating a deadlock at the sending process. This leads to the conclusion that either a message would need to be able to be canceled or the alive-status would need to be requested directly from the receiving process - which would make the coordinating master process dispensable.

On the other hand every process in the network would have to synchronize with the coordinating master process prior to every single communication operation - including collective operations - which would lead to a bottleneck at the master process.

Conclusion

+ Simple approach, quickly implemented
– Consistency problems lead to more complicated implementation
– Approach not scaling in network size; leads to performance issues in large networks

5.5.2 Use topology of Combigrid-Framework

Instead of simulating single process faults, the manager process calculates the probability that a process is dead in a specific time span. For every group, the manager runs a random number generator and decides depending on the quantity of processes present in each group, if a group is to be removed.

Let's assume the probability of a process going dead within a time range $\Delta T > 0$ is $p \in (0, 1)$ and for j processes in the network, for $i \in [0, j-1]$ let $X_{i,\Delta t}(\omega) : \omega \mapsto$

{false, true} be independent random variables indicating if process i is dead after the time Δt.
Furthermore we assume each group consists of k processes and $j = m*k$ for $m \in \mathbb{N}$. Then we have

$$\mathbb{P}(X_{i,\Delta t} = \text{true}) = 1 - (1-p)^{\frac{\Delta t}{\Delta T}} \tag{5.1}$$

and the probability of any process in a specific group is dead after the time Δt is

$$\mathbb{P}(\text{a process in a group has died in the time } \Delta t) = 1 - (1 - \mathbb{P}(X_{i,\Delta t} = \text{true}))^k.$$

For an implementation this means after the passed time Δt, we generate a random value $n \in [0, 1)$ using a uniform pseudo random number generator. Then we calculate the probability p that any process is dead in a given group. If $n < p$, the currently considered group is deleted / dropped.
Note that in our model a once dead process stays dead, as well as a once removed group is also removed permanently.

Example

Consider a network of 129 processes, one manager process and 8 groups containing 16 processes each. Let the probability p of an arbitrary process being dead within one day be $p = 0.001$.
Then the probability of any dead process in a group after a time of $\Delta t = 2$ hours would be

$$1 - (1 - (1 - (1 - 0.001)^{\frac{2}{24}}))^{16} \approx 0.0013$$

or after a time of $\Delta t = 3$ days the probability would be

$$1 - (1 - (1 - (1 - 0.001)^3))^{16} \approx 0.047 \quad .$$

Conclusion

+ Simple approach; very quickly implemented
+ No performance loss, because no MPI functions have to be overwritten by layer functions
+/– Assuming a constant probability of a process failure in a given time range is also not realistic. The reason for a process fault is often due to a hardware failure, for example HDD, RAM or CPU failure and thus the probability of a failure changes with the age of the hardware. If we however only simulate for some hours or days it would be totally fine to use a constant failure rate, because the rate wouldn't change significantly in such a short time.
– Current model assumes the random variables are independent, which is not very realistic due to interactions of processes (for example when a process dies because of overheat, it is much more likely that more processes die as well shortly after it)
– The event 'a process changes its state from alive to dead' can occur in any infinitesimally small time range. So if a process is considered dead, it might be in fact already dead for several time, while it actually went on with its execution.
– This approach does not offer any of the new ULFM features (like MPI_COMM_SHRINK). If the communicator needs to be shrunk, this has to be implemented at the application layer.

5.5.3 Replace all blocking operations with their non-blocking versions

In this approach every blocking collective function is substituted by the corresponding non-blocking collective function. The most simple case is with no present process faults. In this case, all ranks in the communicator will eventually call the collective function. The inside of the layer then called non-blocking collective will afterwards be completed by MPI_TEST or MPI_WAIT to achieve the blocking characteristic.
In case of a present process failure, there are some steps to perform. For a better illustration let's assume the current ongoing blocking collective is MPI_BARRIER.

Step 1: After calling 'kill_me', a dead process sends a notification ("dead-message") to the root rank of the communicator.
Step 2: Once the root rank receives the message, it notifies the other ranks in the communicator about the present dead process and responds to the dead process with a request to begin an MPI_IBARRIER immediately and unblock all other processes.

This approach is similar to the proposed handling of non-blocking collectives. If a dead process is detected, any outstanding operation should eventually be completed to free resources.

The approach only works if the MPI implementation fully supports non-blocking collectives, this is also the main reason why it wasn't used in our implementation. As mentioned at the beginning of this document, non-blocking collectives were introduced in 2012 and some implementations still do not fully support non-blocking collectives.

5.5.4 Using remote procedure call

This approach is more theoretical, because MPI itself does not support interrupts or remote procedure call (abbreviated to RPC) and using functions outside MPI is not the intention of this thesis. Nevertheless this approach possibly offers higher performance.

The idea is that MPI communication functions are called much more frequently than a process dies in a communicator, therefore it could be better to instantly execute MPI functions when called by the application and "clean up the mess" in case a process is detected dead afterwards. While in our previous approaches (including the implemented version) a process was dead the moment it called the "kill me" function. If a process is to be killed in this approach, it calls the "kill me" function and every other process in every currently active communicator is notified via RPC (possibly via some kind of broadcast for better performance) of its intention by a message 'can I kill myself?'. If another process receives this message, it either responds with a message 'ok' if there is currently no outstanding MPI communication with the dead process, or 'not ok' - initializing a completion

protocol to clean up outstanding communication.

5.5.5 Spawning processes or threads

In our implementation the biggest problem is the exchange of information in the background. For our custom non-blocking collectives implementation every process needs to frequently call MPI functions in order to allow the background messages to be transfered.
We could avoid this problem by spawning either a thread or a process for every single MPI process present in MPI_COMM_WORLD. Let's take a look at the case where we spawn new processes (assuming the MPI implementation supports it).
Instead of directly exchanging information about dead processes, a process only contacts its "own" spawned process. The spawned process then can either initiate a background operation or respond with the information that another process is dead, the communicator is revoked, etc.

Also the spawned process does only interact with the parent process (that is the process, which spawned the new process) or with other spawned processes. While this approach seems promising, it has some downsides. Spawning threads or processes well depends on implementation and some implementations don't support spawning threads at all. When threads or spawned processes wait for incoming messages using MPI_RECV, they do not idle the processor but use 100% CPU and thus leads to a performance loss for the application.

For illustration figure 5.24 shows a simplified depiction of this approach. Figure 5.25 shows a simple example using a combined version of spawned processes and the method from approach 5.5.4 (RPC, "clean up the mess").

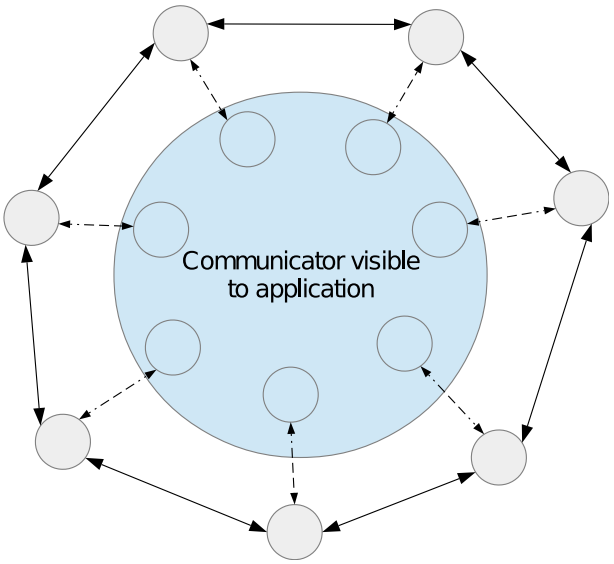
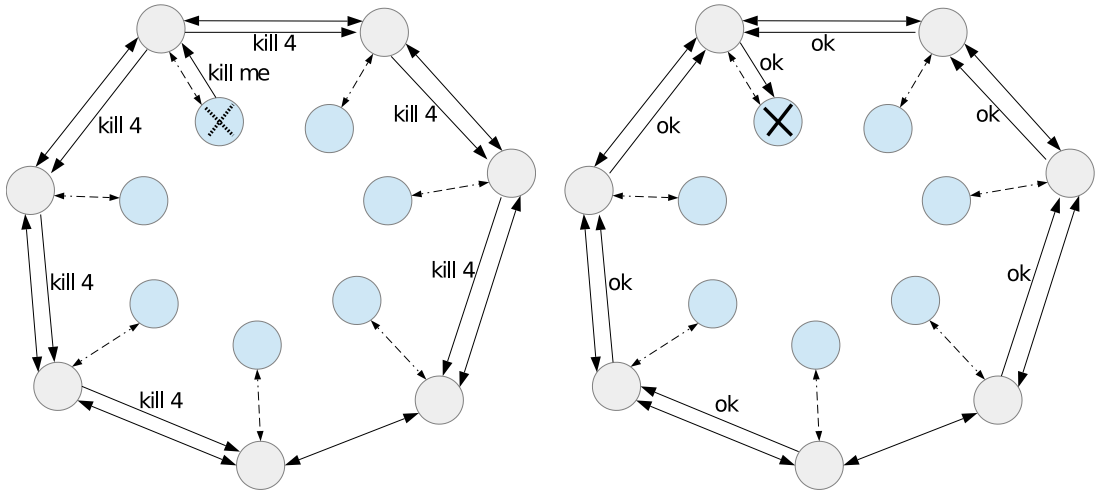


Figure 5.24: Every process has its own spawned partner process outside the communicator visible to the application.



(a) Process 4 sends a kill request to its partner process. The partner process forwards the request to all other partner processes.

(b) In this case no communication with process 4 is active. All partner processes respond with 'ok'. Process 4 receives the 'ok' from its partner process.

Figure 5.25: Example where process 4 wants to be dead.

Conclusion

+ Effectively send background messages without interfering with the application
– Performance and availability heavily depending on the MPI implementation; 100 % CPU usage while waiting for new instructions / messages
– One process or thread is required for each node in the network; needs more resources

5.6 Using the fault simulator

The fault simulator can be integrated into the application by including MPI-FT.h and MPI-FT_redefine.h instead of MPI.h. In MPI-FT.h, the location and name of the regular MPI header can be set in REAL_MPI_INCLUDE, the simulated timeout can be set in SIM_FT_TIMEOUT and the width of our custom tree can be set in TREE_SUCCESSOR_COUNT (where '2' would result in a binary tree).

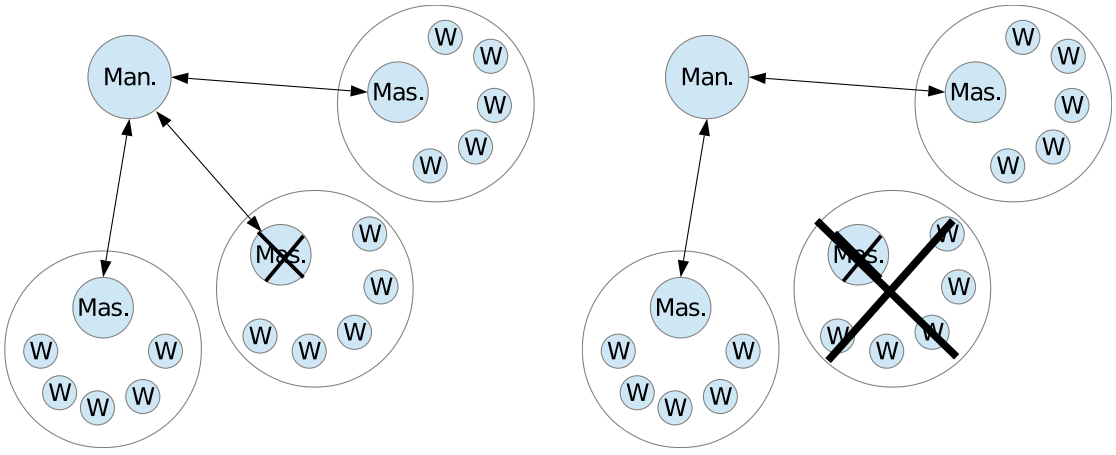
The makefile can be used to create a test program. 'make MPI_Test_ft' compiles with active fault simulator, 'make MPI_Test_regular' compiles without the fault simulator.

6 Integration into Combigrid and tests

6.1 Fault tolerant master/worker

This section is about a possible way of making use of the new ULFM functions to make the current combi technique framework fault tolerant. In the current version of the framework, the manager sends jobs to an arbitrary amount of groups of workers. The more groups available, the faster the computation goes, but not all groups are required for the computation to be functional. We can use this and let the manager process remove every group with a present dead process if detected. The only problem is, how to detect these dead processes.

If a master process is dead (also see Figure 6.1), the manager process will be notified of the dead master process after calling a collective operation followed by an MPI_COMM_AGREE. However, if no collective operation is planned, the manager can also make use of the fact, that an MPI_IRECV with source MPI_ANY_SOURCE will eventually fail with the error MPI_ERR_PROC_PENDING in case of a present dead process.



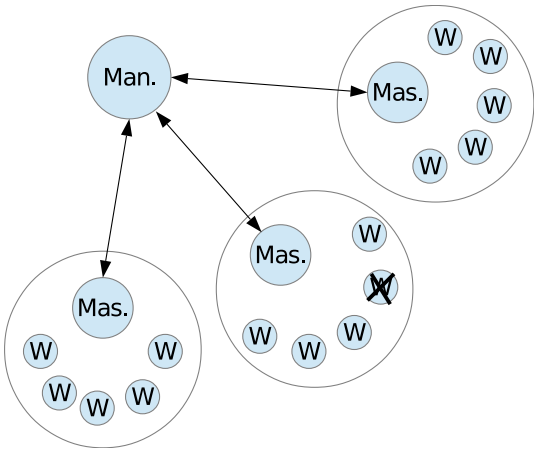
(a) A master process of a group is dead.

(b) The manager gets notified by ULFM about the present dead master process and removes the whole group. No more jobs are sent to that group.

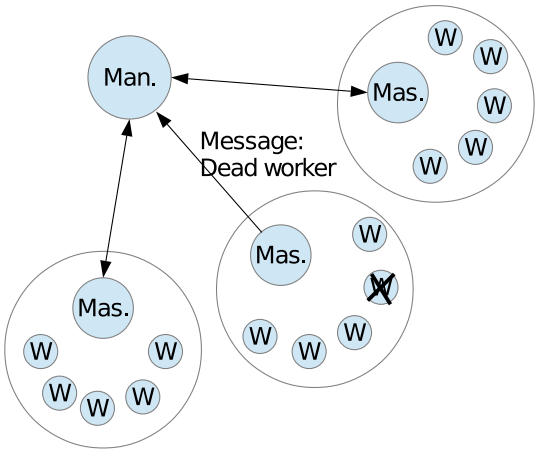
Figure 6.1: Case with a dead master process.

If a worker process is dead (Figure 6.2), we have two possible ways of detecting the dead process at the manager process.
One way would be for the master of the group of the dead worker to send a signal message like "Dead worker" after detecting it. The manager would receive the signal and remove the group.
Another way would be to initiate an MPI_IRECV with source MPI_ANY_SOURCE at the communicator MPI_COMM_WORLD using a tag that is not used anywhere else in the application. This way the manager process can check the request from time to time using MPI_TEST. If then a worker (or master) process is dead, the function will return an error MPI_ERR_PROC_FAILED and we can extract the failed process using MPI_COMM_FAILURE_GET_ACKED after calling MPI_COMM_FAILURE_ACK. After that we have to find the group containing the dead process and delete it. We can use our group of failed processes and intersect it with every other group currently known to the manager using MPI_GROUP_INTERSECTION. If one of the intersected groups has a size > 0 (extracted using MPI_GROUP_SIZE), we know there is a dead process inside and we can remove the corresponding working group.

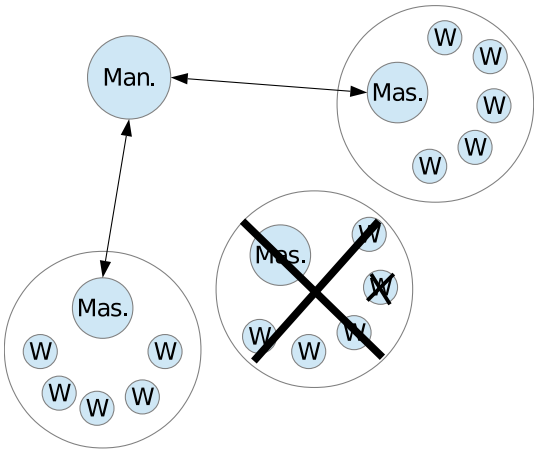
Additionally in our fault simulator it is possible to access the currently dead processes directly by accessing comm–>dead_set or comm–>dead_nodes.



(a) A worker process is dead.



(b) The master of the group gets notified by ULFM about the present dead worker process and sends a status message to the manager, informing it about the present dead worker.



(c) The manager process removes the whole group. No more jobs are sent to that group.

Figure 6.2: Case with a dead worker process.

6.2 Tests

Just like any other software the prototype must be tested to minimize the possibility of software bugs or unwanted behavior. For this purpose we define some test cases. On the one hand we have some functional and stability tests to show that the prototype behaves as intended. On the other hand we define some performance tests and run them with and without the active fault simulator and compare the results, which hopefully are as much identical as possible. While performance is quite important, scalability is even more important, because it should be possible to use the fault simulator with a huge MPI network.

Each test is executed 128 times with n nodes in the MPI network, sending messages of size k integers (with n and k variable sizes, depending on the test). The tests are executed on the communicator MPI_COMM_WORLD. Every test is expected to run without any unexplainable abort or error. The test results are located in the Appendix A.

6.2.1 Performance tests

The following tests shall show, how the use of the fault simulator impacts the performance of an application with no present simulated dead processes. The tests are executed both using a regular MPI only and with the active fault simulator.

Test 1

Each process executes an MPI_ALLREDUCE with root rank 0.

Test 2

$\frac{n}{2}$ processes execute an MPI_SEND, the other $\frac{n}{2}$ processes execute an MPI_RECV so that we have $\frac{n}{2}$ matching MPI_SEND and MPI_RECV.

Test 3

$\frac{n}{2}$ processes execute a (non-blocking) MPI_ISEND, the other $\frac{n}{2}$ processes execute an MPI_IRECV so that we have $\frac{n}{2}$ matching MPI_ISEND and MPI_IRECV. The non-blocking communications are completed by an MPI_WAIT.

Test 4

Each process executes an MPI_BCAST with root rank 0. This test is intended to evince disadvantages of our fault simulator due to the barrier-like synchronization of dead processes prior to a blocking collective.

Test 5

First execute Test1, afterwards execute Test2. This test shows the impact of alternately executing collective and point-to-point operations on the performance and stability.

6.2.2 Stability tests

Test 6

Execute Test1 to Test6 with $j < n$ processes dead.
In the fault layer, the root of a communicator has an important role. The test therefore shall cover a case with root alive (and some other processes dead) and a case with root dead.

Test 7

Execute Test1 multiple times with root and multiple other processes dead. At the end of the test, there should be no memory leaks. The reason for this test is to show correct behavior of the layer in case there are dead processes (especially if root is dead).

Test 8

Create a new communicator with MPI_COMM_SPLIT and free it with MPI_COMM_FREE. Execute this test multiple times. There should be no memory leaks or other unexpected behavior.

6.2.3 Functional tests

The tests in this section shall show the correctness of the simulated faults layer in terms of functionality. These tests are only executed with active fault layer - obviously because we cannot ”kill” a process in the regular MPI. Because basic functionality was already tested in the other tests until this point, we will not test those again.

Test 9

Step 1: In a communicator with present dead processes, create a new communicator using MPI_COMM_SHRINK.
Step 2: On the newly created communicator, execute an MPI_BCAST with specific values to broadcast. The broadcast should execute correctly and all processes should have the correct values received by the broadcast.

Test 10

Revoke an active communicator using MPI_COMM_REVOKE and execute some MPI functions afterwards. The behavior should be in accordance with the ULFM standard.

Test 11

Run Test 8 with a dead process after initiating a non-blocking barrier and broadcast, without completing them. The uncompleted MPI operations shall be successfully completed by the completion protocol.

6.2.4 Test results

We have a big amount of test results and diagrams. In this section we give a brief overview of the most important results. The remaining test results are located in Appendix A.

We run our tests on the HLRS supercomputer Hazel Hen. Because the network performance largely depends on the network usage of other concurrent running applications on the supercomputer, our test results vary significantly and therefore we run the tests multiple times at different times of the day.

An example for this problem: one test run resulted in an overall higher performance with the active fault layer than without the active fault layer - this should not be possible, because the fault layer uses additional resources to the normal MPI functions and the expected run-time is longer than the version without active fault layer. Therefore this test result was discarded.
For authenticity test results were not discarded if only a few values seemed ”odd”.

We run our stability and performance tests with different amounts of processes used, namely 264, 768, 1536, 6144, 12288 and 36864. In our diagrams the x-axis denotes the size k of integers used for the test, the y-axis denotes the time needed for the test to finish. The blue curve illustrates the results having the fault layer active, whereas the red curve only uses the regular MPI without the fault layer.

Performance tests

In Figure 6.3 we take a look at a single test run from Test 1 using 6144 processes. The diagram of our test results contains information about how much time our allreduce operation took to complete (128 times). The scales are logarithmic. The x-axis denotes the amount of integer values transferred in each allreduce operation, the y-axis denotes the time needed to complete 128 of them. The blue curve belongs to the values obtained with the active fault simulator (but without present process failures), the red curve belongs to the ones without active fault layer. The values are as expected. As you can see for small message sizes, the fault layer has (relatively speaking) a high impact on the performance, but the speed is still

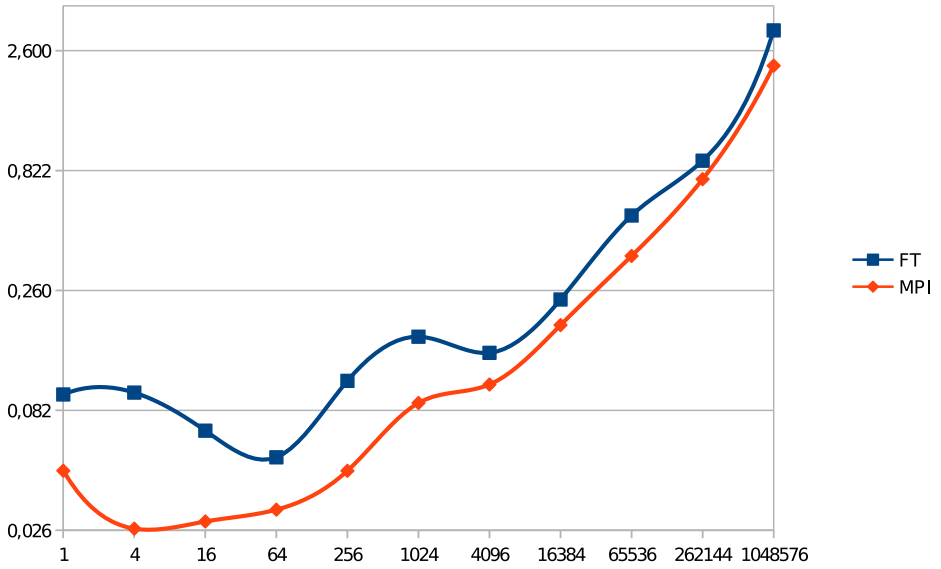


Figure 6.3: 128 allreduce operations using 6144 processes.

acceptable. With growing message size, the relative impact becomes smaller. In Figure 6.4 you can see the same values, but the fault tolerant version relative to the normal MPI version (for example '2' means, the fault simulator version took twice as long).

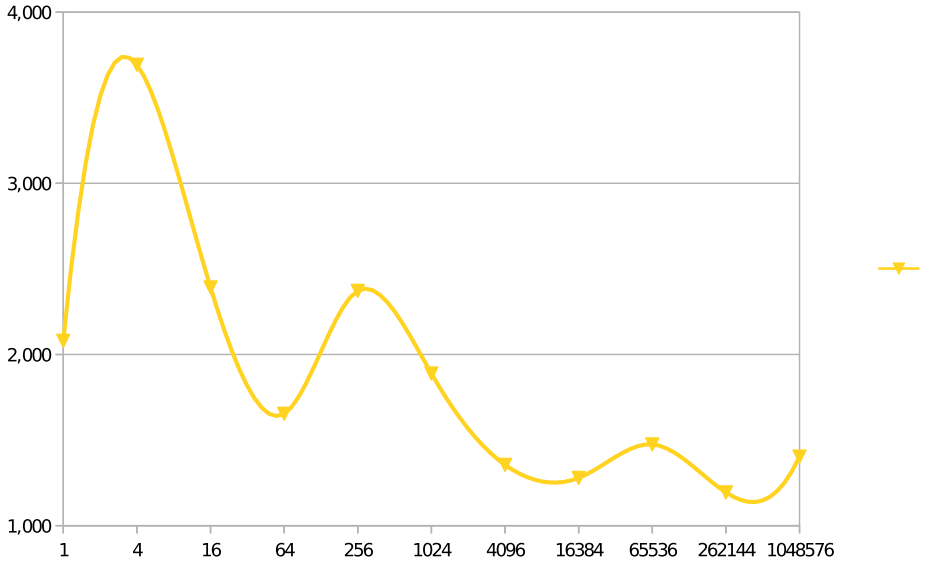


Figure 6.4: 128 allreduce operations using 6144 processes. The curve shows, how much the active fault simulator needed relative to the "regular" MPI version.

Test 4 (test with broadcasts, see diagrams in Appendix A.1.4) has a significant worse result for the fault layer compared to the normal MPI version, this is because our synchronization call Sim_FT_Check_dead_processes (section 5.3.2), behaves like a barrier. Only after all processes have called the function, it is possible to continue with the real MPI_BCAST. Without the fault layer, MPI has the ability to immediately return successfully from the call MPI_BCAST, when the broadcast message is forwarded. After the return, the next MPI_BCAST call is initialized (as the 128 operations are called in a loop). So if an MPI_BCAST is not even finished at all ranks, one or more MPI_BCAST operations could already be initiated - even though the calls are blocking - and therefore gain a significant performance boost compared to our fault layer. This is one of the bigger disadvantages of our fault layer. Although the performance is still acceptable, some MPI optimizations aren't possible anymore with the active fault simulator, especially using blocking operations.

The other test results concerning performance tests are located in the Appendix A.

Stability tests and functional tests

For the stability tests, we have no diagrams, because the tests require present dead processes and therefore a comparison with the "regular" MPI version was not possible. The tests were also run on the Hazel Hen and performed well, the behavior was as expected. The test program MPI_Test is included in the source code.

6.3 Conclusion and outlook

Our implementation of a fault simulator allows the simulation of a possible ULFM implementation by using any recent regular MPI implementation. The most important ULFM functions, needed for the integration in the combi technique framework, were implemented.

In our simulator, blocking MPI operations check for present process faults prior to the operation call. Only if all processes participating in the operation are alive, the operation is executed, otherwise an error is returned. Non-blocking operations are called immediately and the "alive-checking" is done in a completion function like MPI_TEST.
Messages within our fault layer are transferred via special duplications of the communicator used by the application layer. This way the messages don't interfere with each other.
In MPI a communicator can only be deleted using MPI_COMM_FREE, if no non-blocking operations are outstanding. We therefore implemented a completion protocol, allowing outstanding non-blocking collectives to be completed at all ranks in the communicator (including the dead ones).

As seen in the test results, our implementation scales logarithmic in network size and can therefore be used in huge MPI networks. For compatibility we did not use any native non-blocking MPI collectives for our fault layer and therefore our fault simulator can be used with MPI implementations using a standard before MPI 3.0.

Outlook

While completion of non-blocking collectives was successfully implemented as a "proof of concept", outstanding non-blocking point-to-point operations are currently ignored in our layer. If their completion is required as well, a similar protocol as with our non-blocking collectives can be implemented. The only difference would be, that any non-blocking point-to-point operation call has to be stored for each used TAG separately.

In the ULFM standard exist some more new functions, not implemented in our simulator. These are for example functions regarding Windows and file access.

Many of our layer functions are independent of ULFM. If in the future a fault simulator using another fault tolerant MPI standard is required, this can be implemented quickly.

Although the current ULFM proposal wasn't integrated into the MPI standard 3.1, it is very likely to get this functionality in future MPI versions. With the existence of real fault tolerance, a fault simulator is not required anymore, a dead process is then achieved by just killing the process with a task manager.

A Test results (full)

In the following part are the remaining results of our test runs involving comparison between the tests with active fault simulator and without the active fault simulator.

A.1 Performance Tests

A.1.1 Test 1

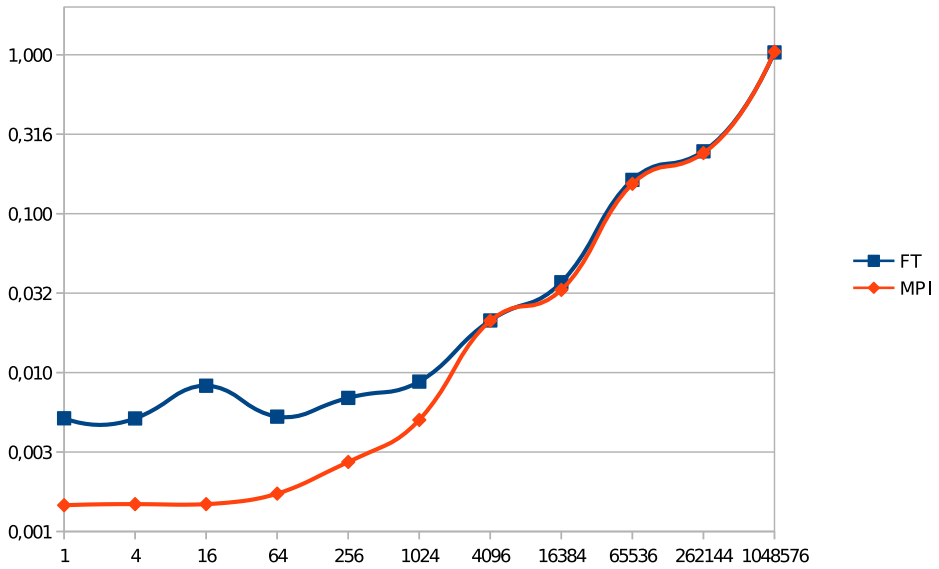


Figure A.1: 128 allreduce operations using 264 processes.

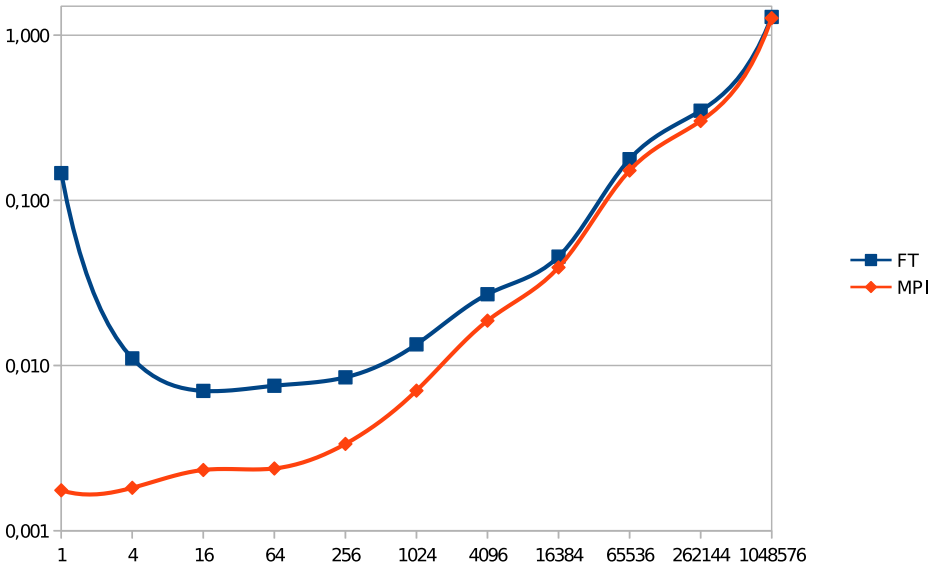


Figure A.2: 128 allreduce operations using 768 processes. There was some network load at the beginning of the fault layer tests resulting in a higher test run time than expected.

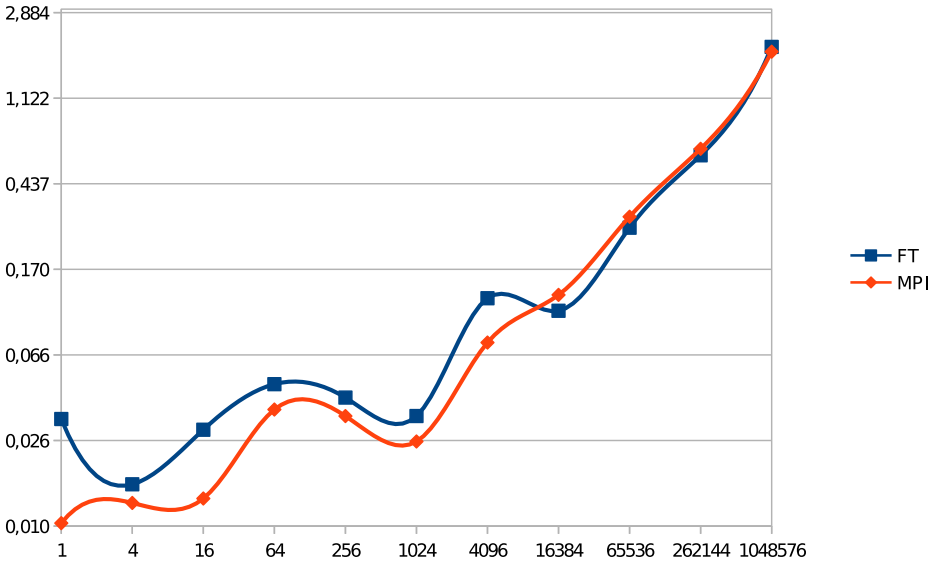


Figure A.3: 128 allreduce operations using 1536 processes.

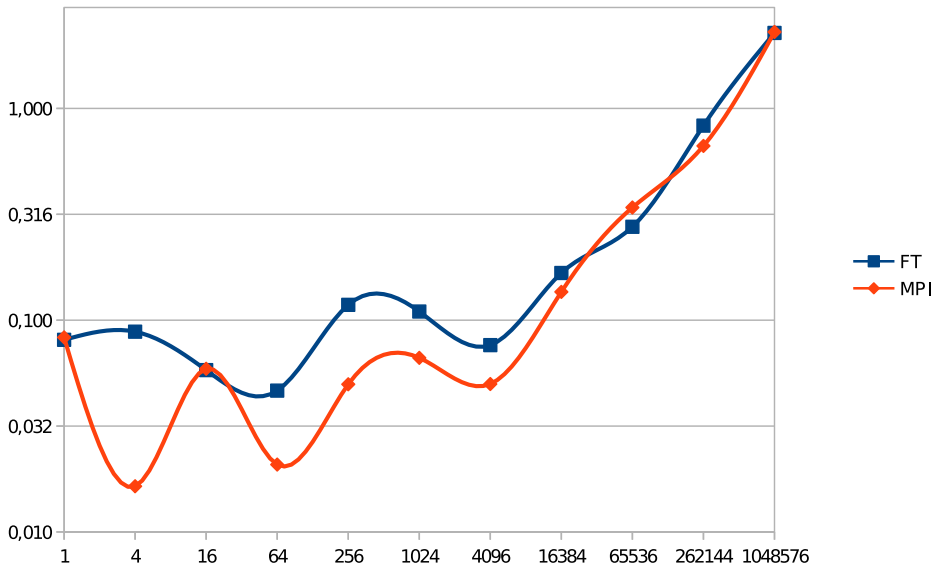


Figure A.4: 128 allreduce operations using 12288 processes.

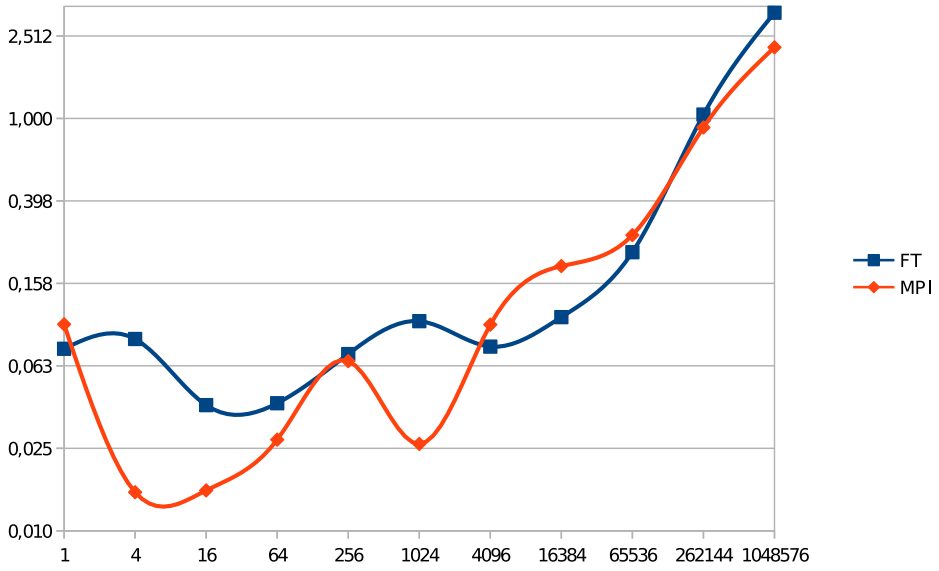


Figure A.5: 128 allreduce operations using 36864 processes.

A.1.2 Test 2

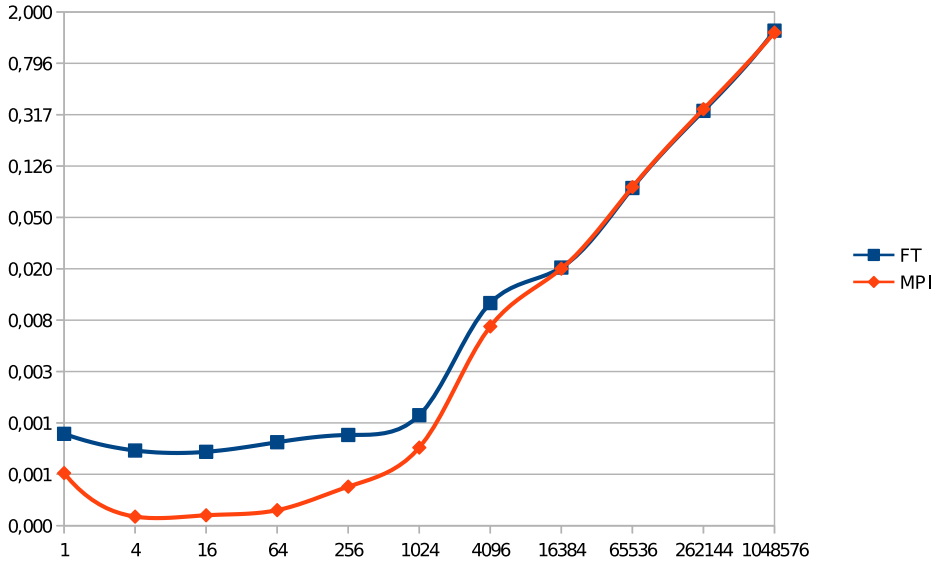


Figure A.6: 128 blocking point-to-point operations using 264 processes.

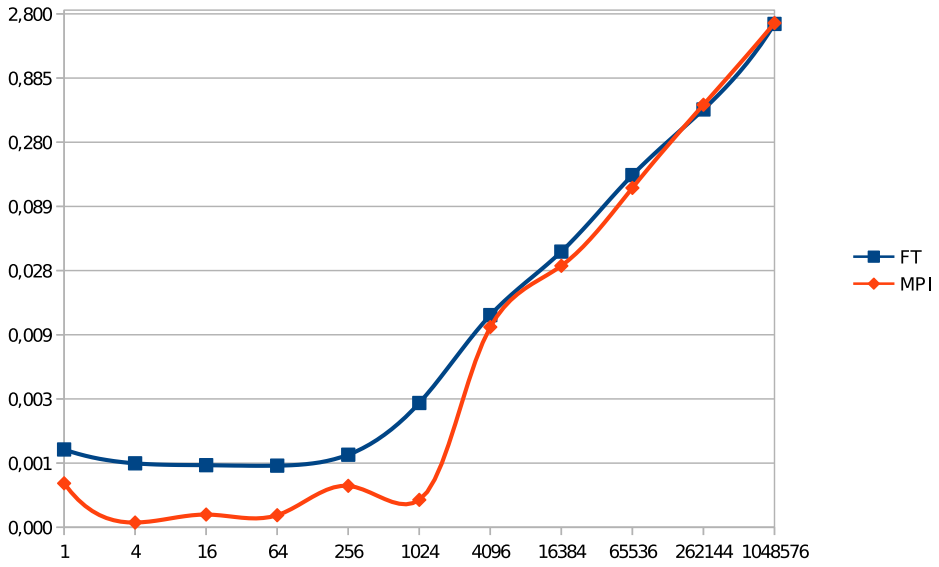


Figure A.7: 128 blocking point-to-point operations using 768 processes.

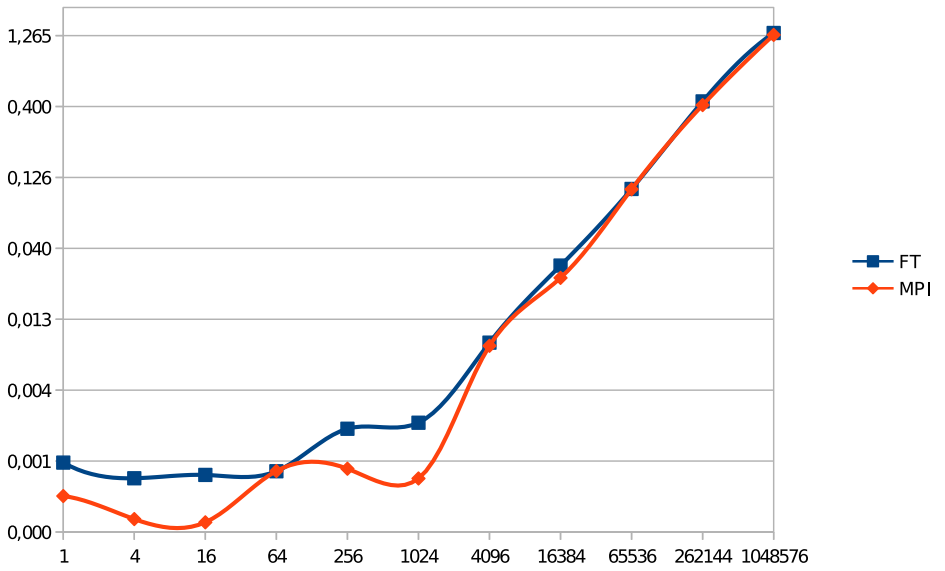


Figure A.8: 128 blocking point-to-point operations using 1536 processes.

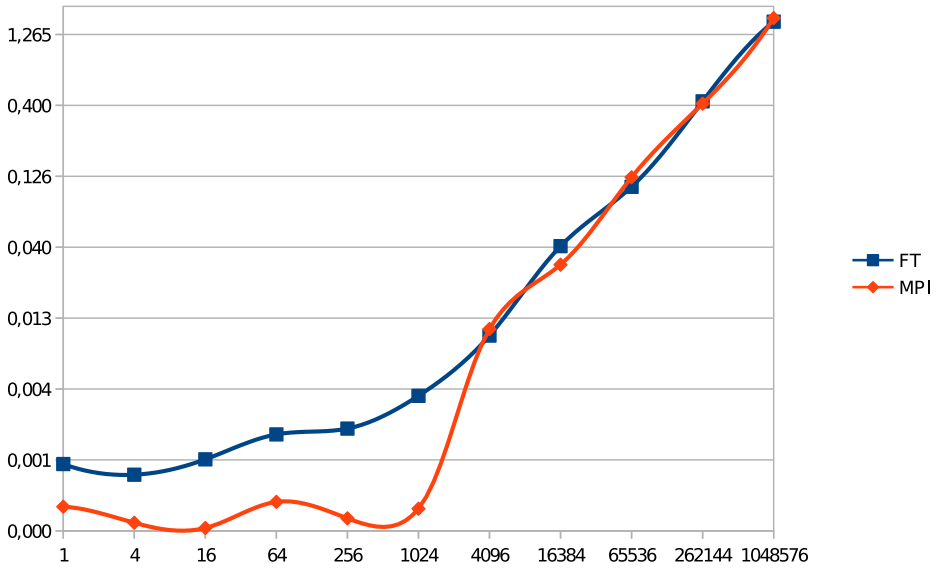


Figure A.9: 128 blocking point-to-point operations using 6144 processes.

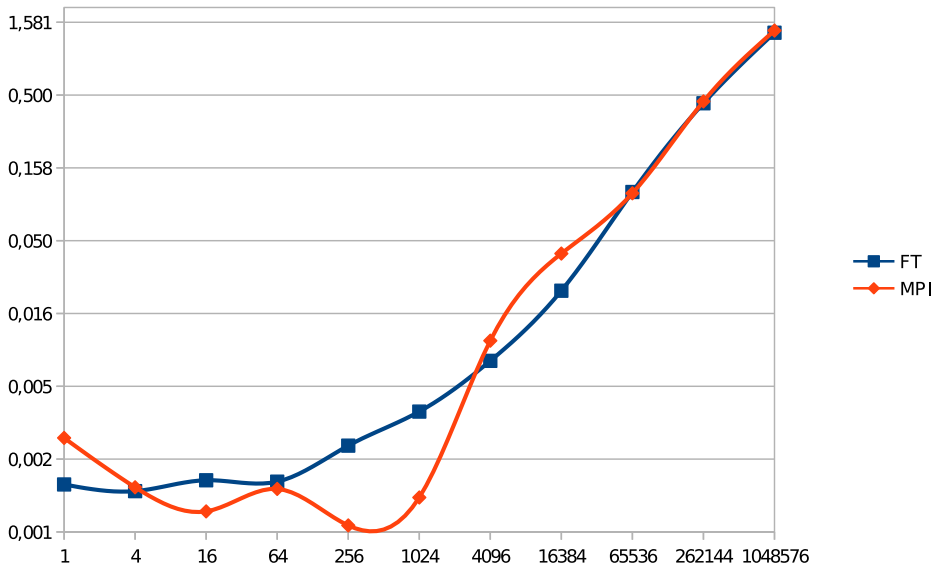


Figure A.10: 128 blocking point-to-point operations using 12288 processes.

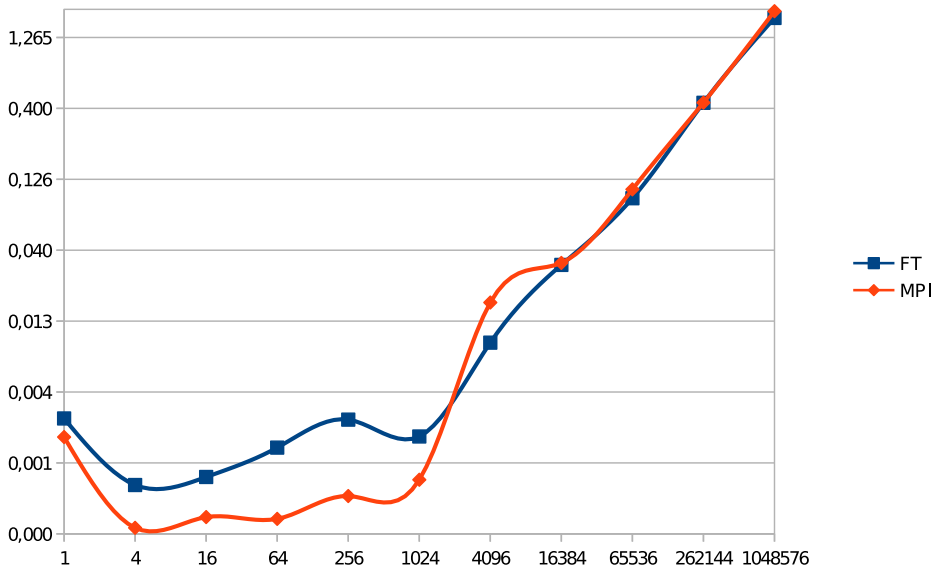


Figure A.11: 128 blocking point-to-point operations using 36864 processes.

A.1.3 Test 3

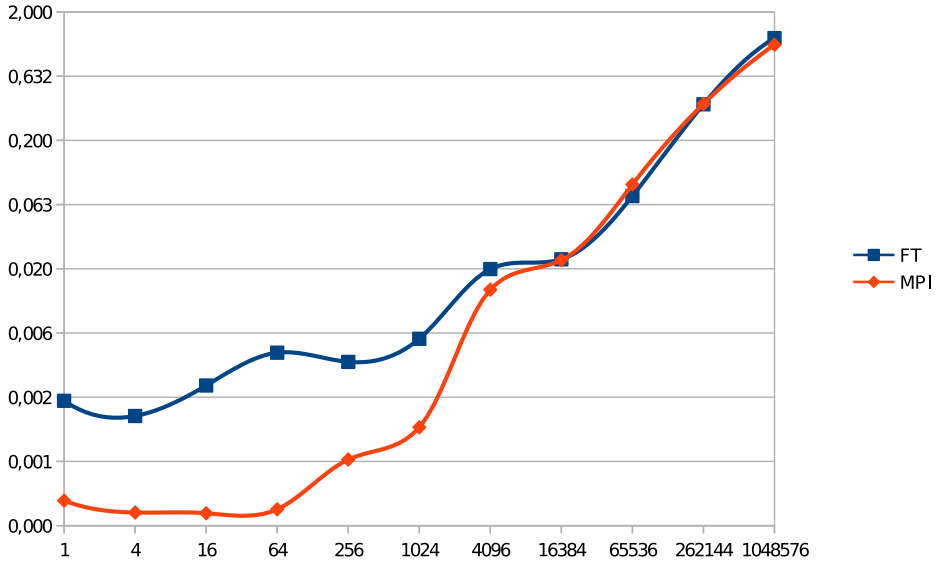


Figure A.12: 128 blocking point-to-point operations using 264 processes.

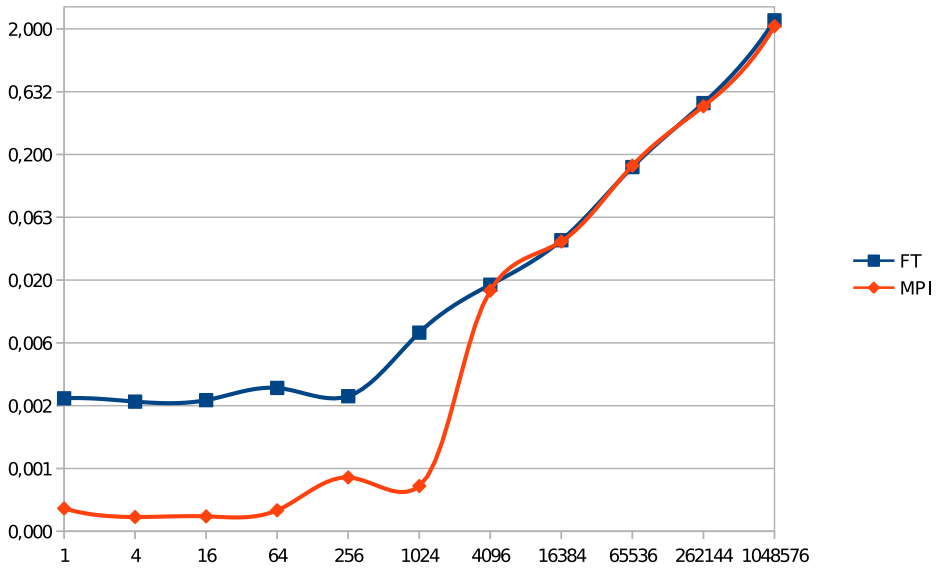


Figure A.13: 128 blocking point-to-point operations using 768 processes.

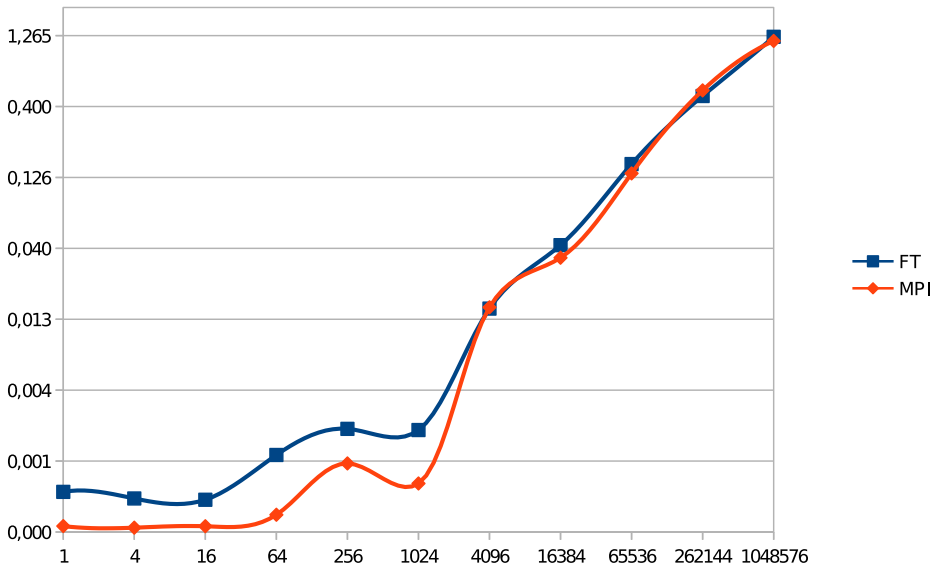


Figure A.14: 128 blocking point-to-point operations using 1536 processes.

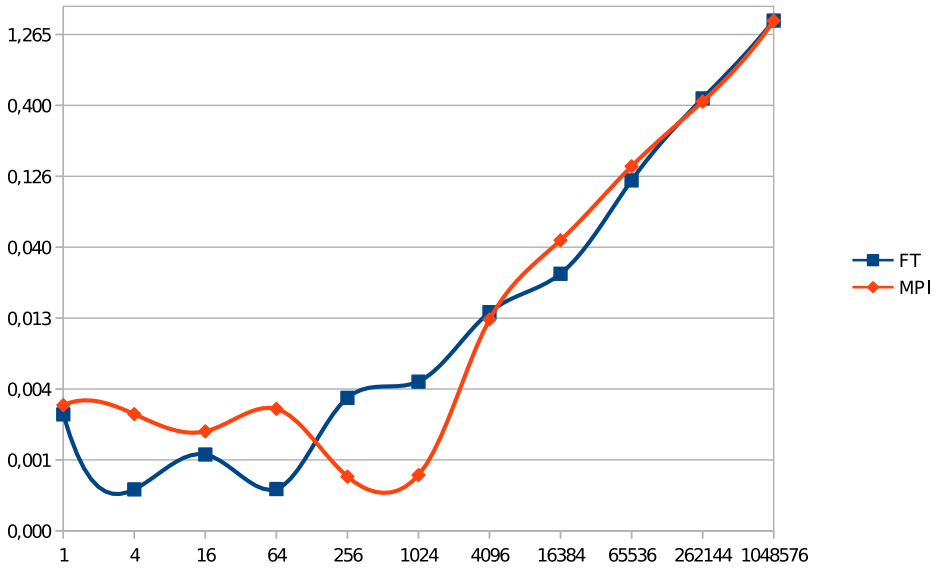


Figure A.15: 128 blocking point-to-point operations using 6144 processes.

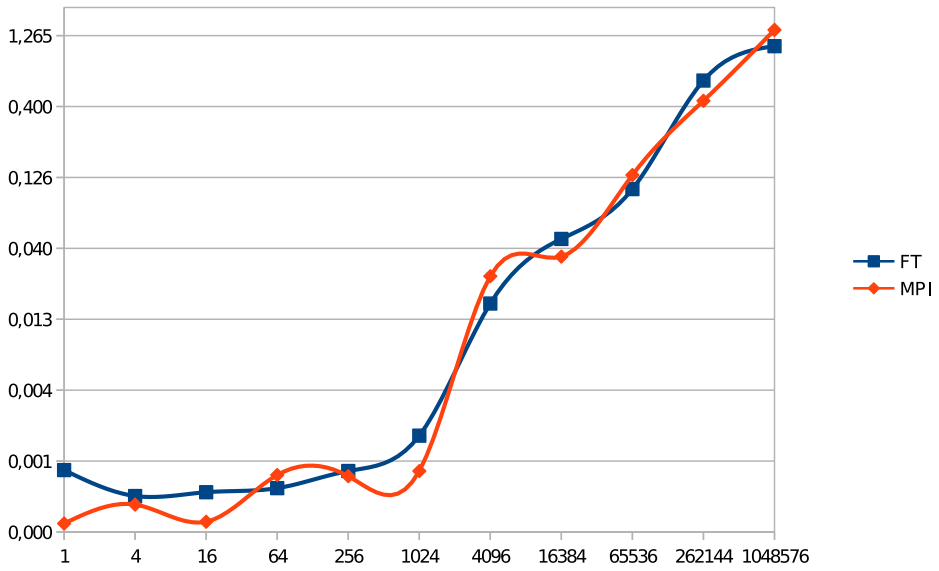


Figure A.16: 128 blocking point-to-point operations using 12288 processes.

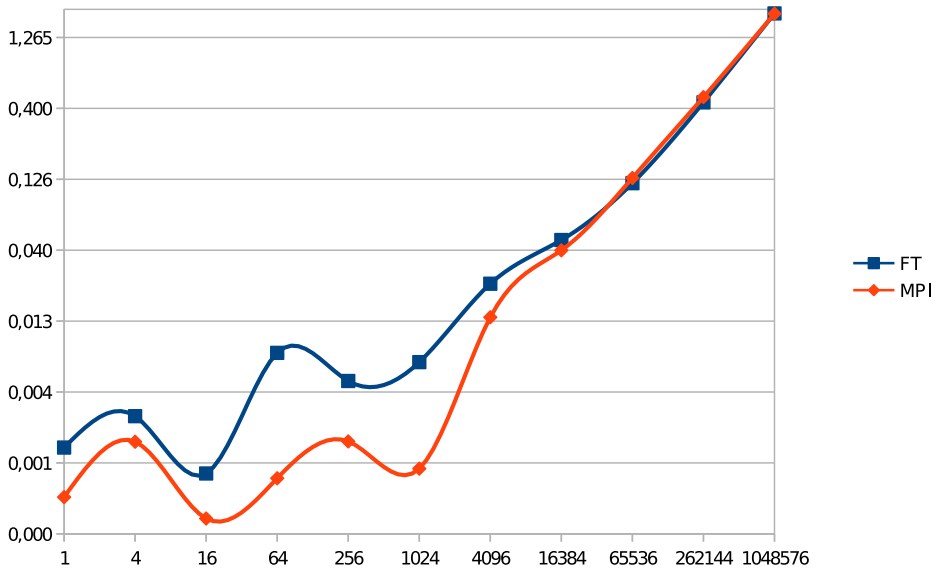


Figure A.17: 128 blocking point-to-point operations using 36864 processes.

A.1.4 Test 4

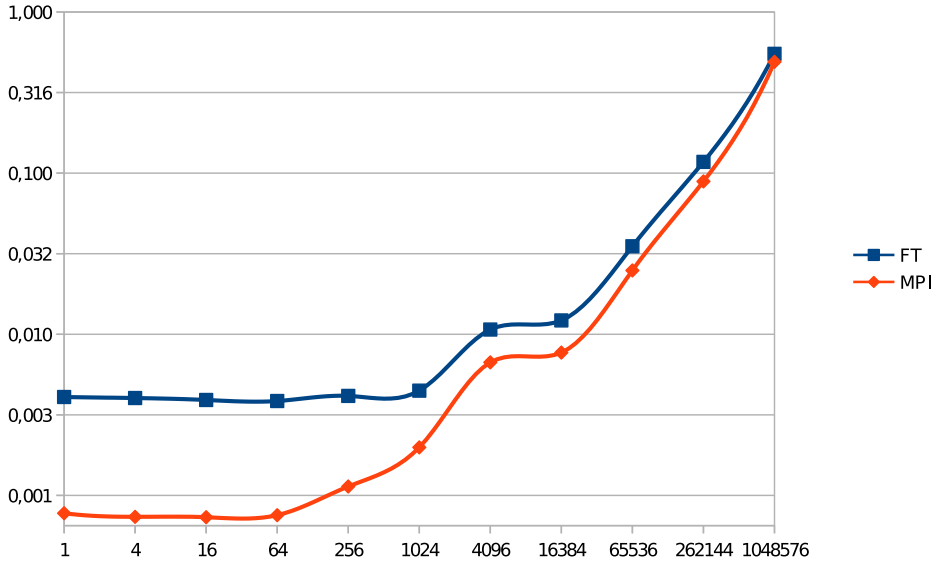


Figure A.18: 128 blocking broadcast operations using 264 processes.

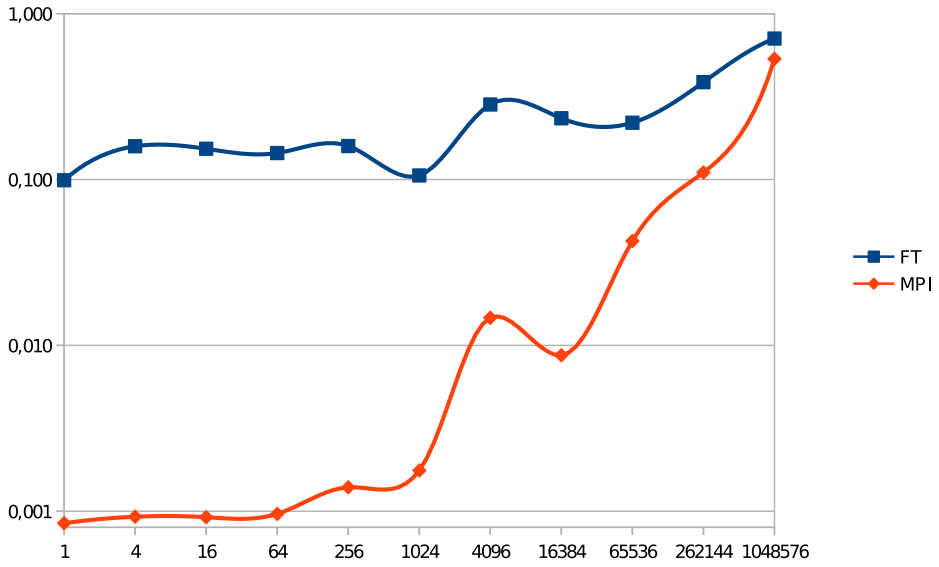


Figure A.19: 128 blocking broadcast operations using 768 processes.

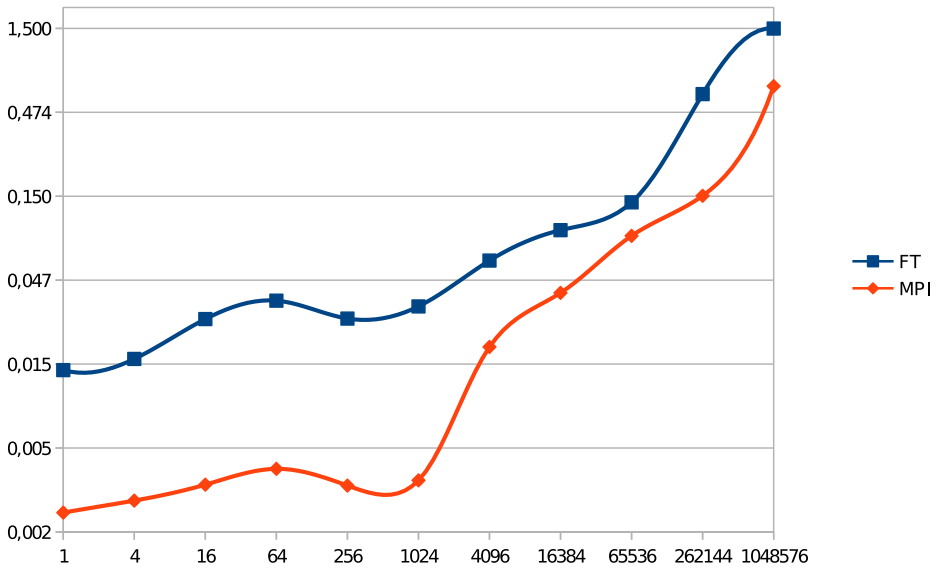


Figure A.20: 128 blocking broadcast operations using 1536 processes.

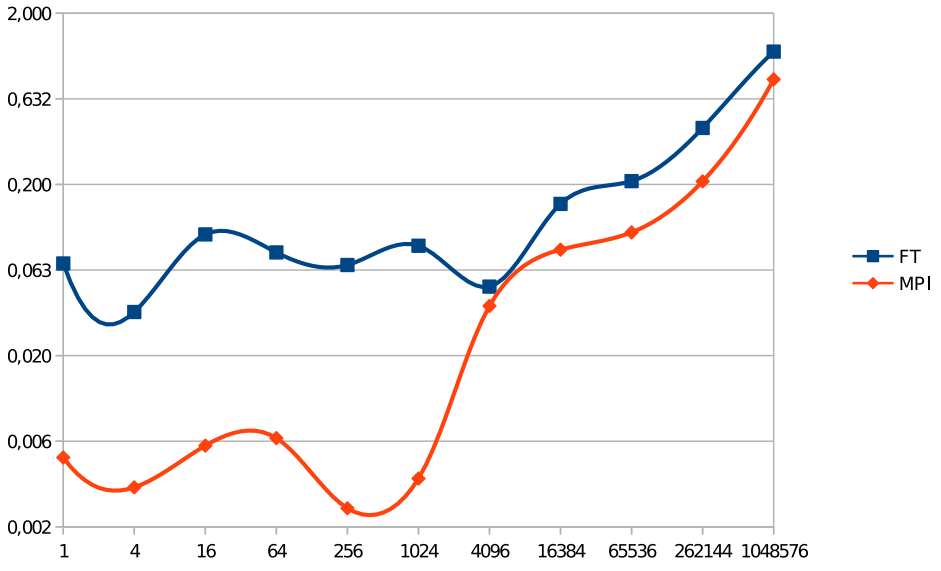


Figure A.21: 128 blocking broadcast operations using 6144 processes.

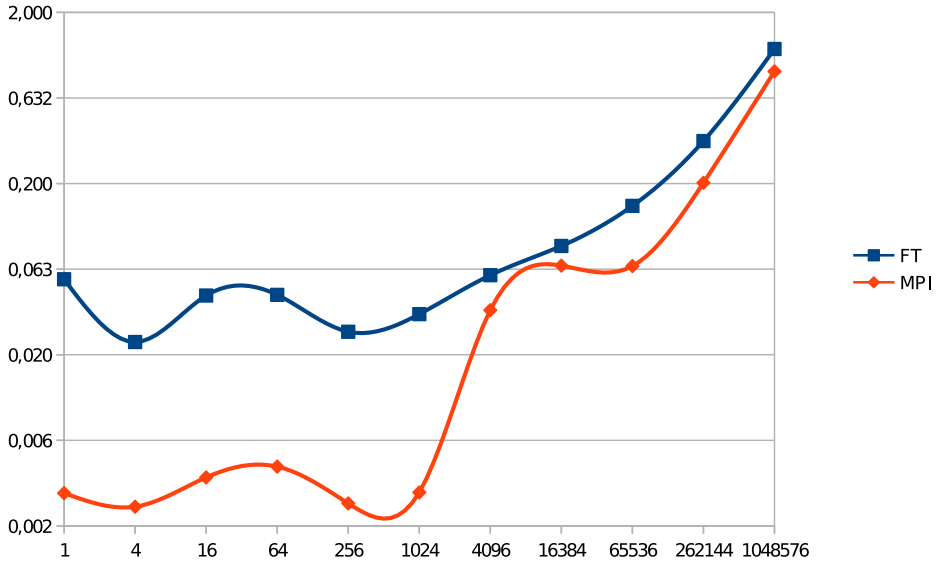


Figure A.22: 128 blocking broadcast operations using 12288 processes.

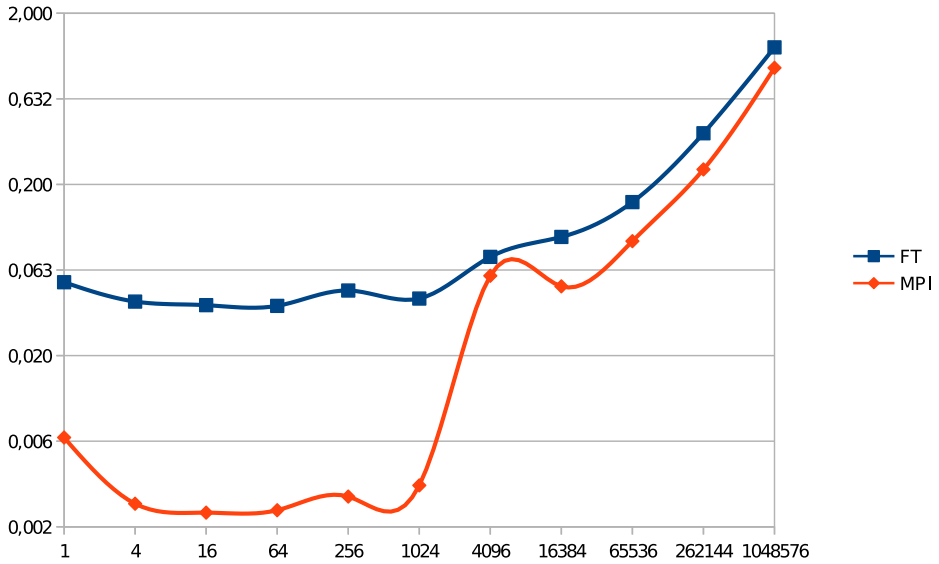


Figure A.23: 128 blocking broadcast operations using 36864 processes.

A.1.5 Test 5

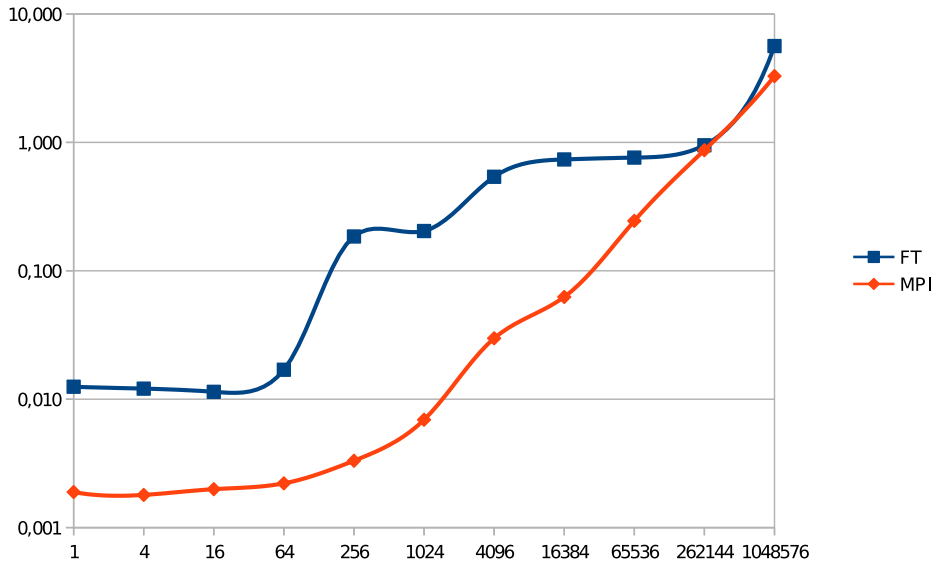


Figure A.24: 128 blocking broadcast operations, combined with blocking point-to-point operations using 264 processes.

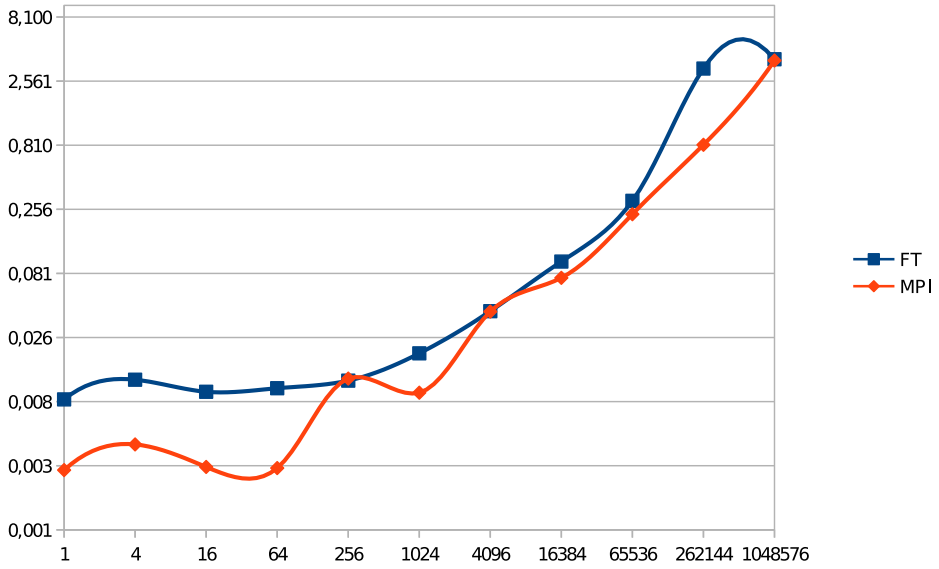


Figure A.25: 128 blocking broadcast operations, combined with blocking point-to-point operations using 768 processes.

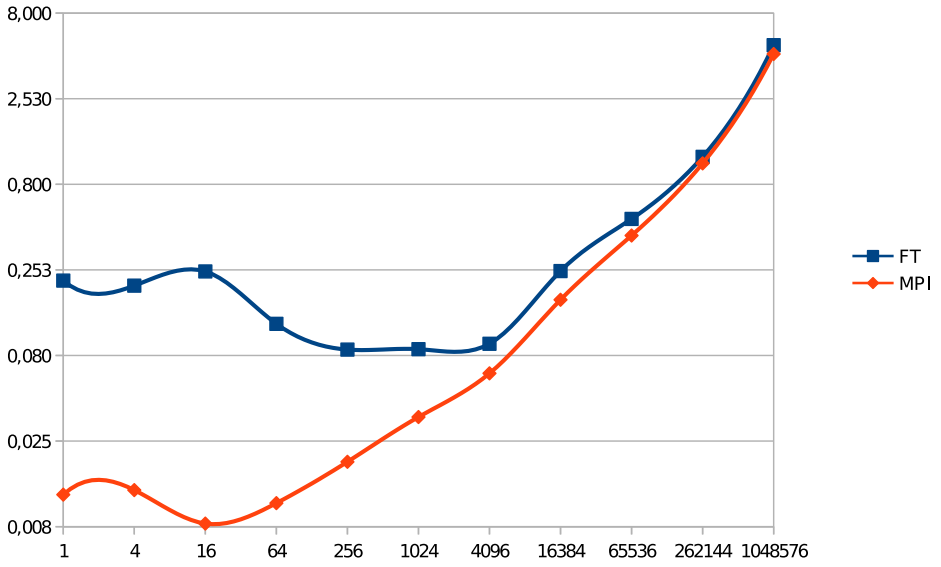


Figure A.26: 128 blocking broadcast operations, combined with blocking point-to-point operations using 1536 processes.

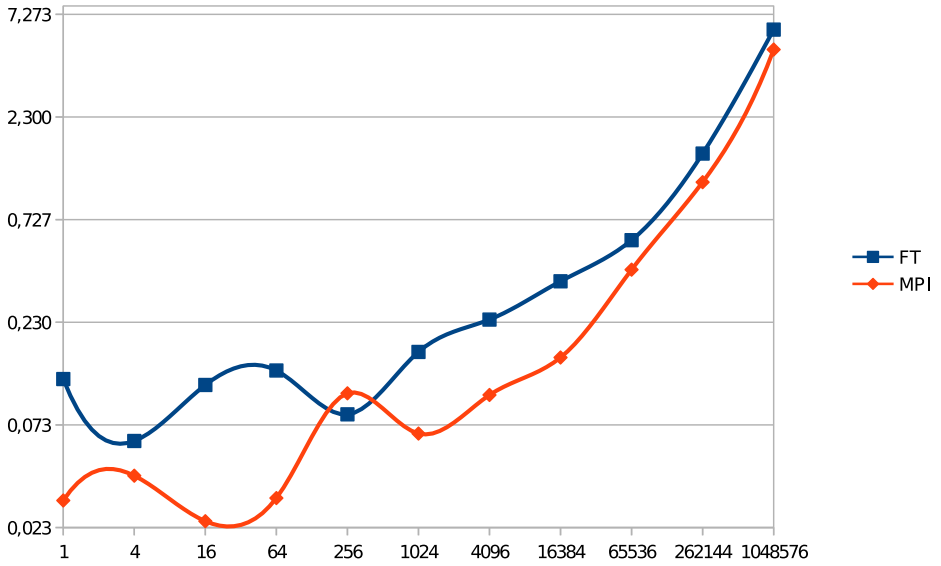


Figure A.27: 128 blocking broadcast operations, combined with blocking point-to-point operations using 6144 processes.

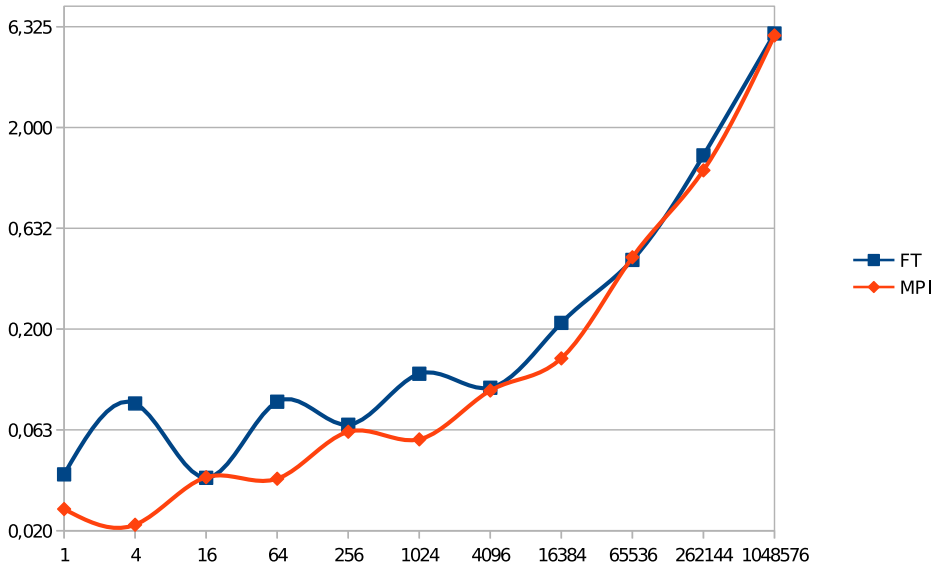


Figure A.28: 128 blocking broadcast operations, combined with blocking point-to-point operations using 12288 processes.

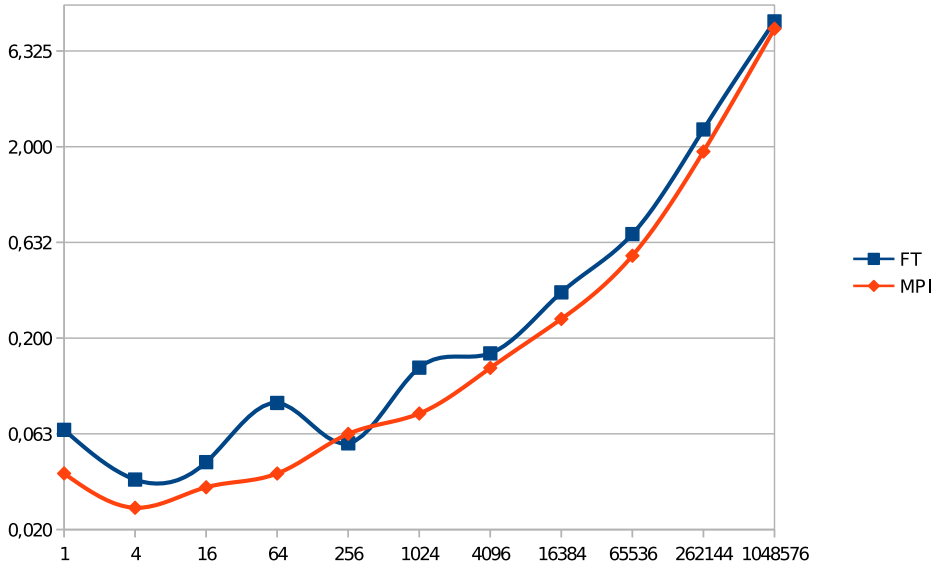


Figure A.29: 128 blocking broadcast operations, combined with blocking point-to-point operations using 36864 processes.

List of Figures

Bibliography

[HHJP] Hupp, Philipp ; Heene, Mario ; Jacob, Riko ; Pflüger, Dirk: *Global communication schemes for the numerical solution of high-dimensional PDEs. Parallel Computing.* Submitted.,

[HKH+15] Hinojosa, Alfredo P. ; Kowitz, Christoph ; Heene, Mario ; Pflüger, Dirk ; Bungartz, Hans-Joachim: Towards a fault-tolerant, scalable implementation of GENE. In: *Proceedings of ICCE 2014*, Springer-Verlag, January 2015 (Lecture Notes in Computational Science and Engineering)

[IEE90] *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Standards Board, (Std 610.12-1990), 1990. ISBN 1–55937467–X

[KKL05] Kola, George ; Kosar, Tevfik ; Livny, Miron: *Faults in Large Distributed Systems and What We Can Do About Them.* Euro-Par'05 Proceedings of the 11th international Euro-Par conference on Parallel Processing Pages 442-453, 2005

[KRS+13] Kemme, Bettina ; Ramalingam, Ganesan ; Schiper, André ; Shapiro, Marc ; Vaswani, Kapil: Consistency in Distributed Systems. In: *Dagstuhl Reports* 3 (2013), Juni, Nr. 2, 92-126. `http://dx.doi.org/10.4230/DagRep.3.2.92`. – DOI 10.4230/DagRep.3.2.92

[MPI15] *MPI: A Message-Passing Interface Standard, Version 3.1.* http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, 2015

[Tan08] Tanenbaum, Andrew S.: *Modern Operating Systems.* Third Edition. Prentice Hall International, 2008. – ISBN 978–0138134594

[ULF] *ULFM: User Level Failure Mitigation.* https://svn.mpi-forum.org/trac/mpi-forum-web/attachment/ticket/323/mpi31-t323-r419-20150301.pdf,

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Unterschrift:

Stuttgart, 11.01.2016

Declaration

I hereby declare that the work presented in this thesis is entirely my own.
I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations.
Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before.
The electronic copy is consistent with all submitted copies.

Signature:

Stuttgart, 11.01.2016