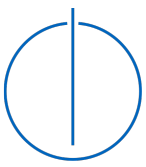
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

# Towards Resilience Methods for Simulation Applications based on Actor Replication

**Manuel Schnaus**



DEPARTMENT OF INFORMATICS

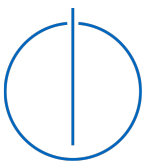
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

# Towards Resilience Methods for Simulation Applications based on Actor Replication

# Aktorenreplikation zur Steigerung der Resilienz von Simulationsanwendungen

| | |
|---|---|
| Author: | Manuel Schnaus |
| Supervisor: | Prof. Dr. Michael Bader |
| Advisor: | Philipp Samfass, Mario Wille |
| Submission Date: | 15.08.2021 |



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 12.08.2021 Manuel Schnaus

# Acknowledgments

Foremost, I would like to thank my advisors, Philipp Samfass and Mario Wille, for their thorough insights into high-performance computing and for always being there for questions. I also would like to thank my family and Kim Le for supporting me during my time writing and for assisting me during the proofreading.

# Abstract

High-performance computing is an important field of scientific computing with many problems offering the possibility of achieving speedups through high levels of parallelization. One framework for programming such a parallelized program is the actor model. This approach establishes the Single Program Multiple Data (SPMD) principle through actors advancing the program and communicating with each other through specified channels. Especially in exascale computing, undetected data corruptions in an actor can have devastating effects on program executions. In order to detect possible data corruptions, I propose to employ double redundancy through full replication of actors. Redundantly computed results can be checked against each other to find errors. Another important task in high-performance computing is balancing the workload evenly between cores. While other approaches achieve promising results on scenarios where imbalances are predictable, they cannot protect the program against non-static and unpredictable imbalances. For these applications, the possibilty of load balancing through redundancy is explored. Here, when an actor is slowed down due to imbalances, its replica can take over and complete the computations, reducing the waiting times of neighboring actors. Using replication, errors within the actor model were observed to be detected with a particularly high accuracy under the sacrifice of runtime. Additionally, the idle time of the actors in unbalanced scenarios was reduced dramatically using load balancing through redundancy.

# Kurzfassung

High-Performance Computing ist ein wichtiges Feld des wissenschaftlichen Rechnens, bei dem sich einige Anwendungen durch Parallelisierung beschleunigen lassen. Ein Framework für solche parallele Programme ist das Aktorenmodell. Dieser Ansatz setzt das Single Program Multiple Data (SPMD) Prinzip ein, indem mehrere Aktoren die Programmausführung übernehmen und während der parallelen Ausführung miteinander kommunizieren. Besonders in Exascale-Anwendungen können unentdeckte Silent Data Corruptions dramatische Auswirkungen auf ein Programm haben. Um solche Fehler zu identifizieren, verwende ich eine doppelte Redundanz über eine Replikation von Aktoren. Redundante Ergebnisse können miteinander verglichen werden, um mögliche Fehler zu finden. Eine weitere wichtige Aufgabe im High-Performance Computing ist die Balancierung der Rechenarbeit zwischen den Rechenkernen. Obwohl andere Ansätze in Szenarien mit vorhersehbaren Ungleichgewichten gute Ergebnisse erzielen, können diese Methoden Programme nicht gegen unvorhersehbare Ungleichgewichte schützen. Für diese Fälle wird die Möglichkeit des Lastausgleiches durch Redundanz vorgeschlagen. Hierbei werden die Berechnungen eines langsamen Aktors von dessen Replika übernommen, wodurch die Wartezeiten der benachbarten Aktoren reduziert werden. Über die Replikation ist es möglich, im Aktorenmodell sehr genau Fehler zu finden, wobei jedoch Laufzeit des Programms geopfert wird. Zudem werden die inaktiven Zeiten von Aktoren in unbalancierten Szenarien dramatisch durch die Lastbalancierung über Redundanz reduziert.

# Contents

# 1. Introduction

With today's algorithms becoming increasingly expensive computationally, achieving a low time-to-solution is an important goal to keep in mind in computer science. For example scientific applications like simulations of real-life scenarios can easily become computationally intense while also being time-sensitive, in which case it is particularly important to find ways of reducing the time needed for program executions. One common way of reducing the runtime is parallelization. Here, the principle of Single Program Multiple Data (SPMD) is often used to execute the same code with different data on multiple computational units [16]. One popular framework for generally implementing parallelism in C++ is UPC++ [22], which offers methods for distributing the execution on multiple ranks and access data across those ranks. UPC++ is a low-level framework, which increases the effort needed for programming. *Actorlib* [19] improves upon this by adding a layer of abstraction offering templates through which the programmer can easily manage the program execution and communication between ranks. This abstraction allows the user to create actors on the ranks which perform all computations in parallel while communicating with each other through specified channels.

While parallelization can undoubtedly heavily increase the speed of an algorithm, it also carries some difficulties. In high-performance settings with many calculations and a large amount of communication between nodes, there is a relevant possibility of errors occuring within the internal processes of a cluster [9]. These errors can happen due to untraceable sources like worn down silicon in the processors and tend to corrupt the memory through bitflips. After a bitflip makes its way into the application, a program can show unexpected behavior like an early failure or even invalid results. The latter can be particularly dangerous if the data corruption goes unnoticed by the program.

Another difficulty of executing a program in parallel is balancing the workload to all available resources. Since each computational unit does the computations on different data and certain nodes may be slower than others, some units might take a longer time for their computations than others. Especially when the computational units communicate with each other, one of them being slowed down might cause the

others having to wait for results.

In order to tackle those two problems in *Actorlib*, the use of double redundancy is explored in this thesis. This redundancy is implemented while still keeping the level of abstraction of *Actorlib* by replicating each actor and therefore executing their programs two times. In this thesis, I demonstrate that replication in *Actorlib* can detect errors and reduce idle times in unbalanced scenarios.

# 2. Related Work

Resilience is an important aspect of computations. Errors are always possible to occur and a program should be able to withstand them to a certain degree. Especially in exascale scenarios, the likelihood of errors increases and the importance of handling them rises with it. There are multiple approaches for handling detected data corruptions during a program execution.

The most common approach is to use a checkpointing system based on regularly storing the state of the program and then using this state as an error-free point of execution from which the program can restart should an error occur. Here, typically the program is stopped at regular intervals and a checkpoint of the current program state is saved to a permanent storage device [7].

The form of the checkpoints written into the permanent storage can be classified into system-level, application-level and runtime-based checkpoints [17]. System-level checkpoints store the entire system information of the machine including the CPU registers and the entire address space of the processes in a checkpoint. In this case, the programmer and the application itself have no information on the checkpoint. In contrast, using application-level checkpoints, the programmer specifies which data should be stored in the checkpoints. As a result, the amount of data needed for each checkpoint is reduced drastically and the programmer can fit the checkpointing closely to the application. Runtime-based checkpointing uses a similar concept to application-level checkpointing. Here, the runtime system provides an interface the programmer can use to implement checkpoints fitting the application while still allowing the runtime system to take over some tasks in the background. Yet, in exascale computing, checkpointing as described here still needs a large amount of memory and computational resources [17].

One improvement to the checkpointing system is multilevel checkpointing [18]. This method introduces multiple levels of resiliency and cost to each checkpoint using the fact that not every failure is equally grave. While the checkpoints with the highest resilience are written into the permanent storage and can withstand a system failure, the checkpoints with a lower resilience can be written into the RAM or other faster

storage. Using this approach, the cost of checkpointing can be reduced significantly.

Another way of handling errors during a program execution is message logging [7]. Here, all messages that a process receives are logged. If an error occurs in one process, it is restarted and the stored messages can be used to quickly advance it to the latest state [13]. This approach can either use a pessimistic or an optimistic method for message logging. The pessimistic approach first blocks all communication of a process when receiving a message. Then, only when the received message is safely logged, the process can continue to send messages. Alternatively, an optimistic approach can be chosen where a received message can be logged later while the process still continues its communication. While this approach can cause an error to cascade, especially in a scenario where failures are rare, an optimistic approach can achieve better performances than the pessimistic method [13].

In order to use these methods, silent errors like bitflips first have to be found during execution. There are multiple approaches that can do so with varying degrees of accuracy and cost [6]. One method to discover errors is *failure prediction* [6]. In order to do this, for every step in the algorithm, an additional predicted numerical value is calculated. This can for example be an approximation for a result of the following timestep in a simulation. Then, a range is calculated surrounding the predicted value. If the computed value is located outside of this area of possible results, an error is detected [3]. When predicting the expected value, it is also possible to determine the impact a possible data corruption could have on a program [8].

A more accurate approach to detect data corruptions is redundancy-based error detection. This method can be split into hardware-based and software-based redundancy. In the hardware-based approach, redundant chips are added to the hardware. During execution, the results of the redundant chips are compared and possible deviations or errors can be found [23]. This type of redundancy is very common in hardware with a high need for resilience. For example in the local computers of cars, redundancy can be used to avoid errors possibly causing accidents [2]. In software-based redundancy, certain computing units are logically replicated on a system or application level and executed multiple times. One example for this is *RedMPI* [10], an extended MPI implementation that duplicates MPI tasks and checks the results for deviances. One possible extension to software-based redundancy is adaptively choosing which resources should be replicated [12].

Alternatively, it is possible to tailor error resilience directly for the executed algorithm [5]. Doing this, the computations in the algorithm are chosen to have numerical

properties inherently robust against silent errors.

In this thesis, an approach using software-based replication is chosen. Here, similarly to *RedMPI*, code is executed two times using the redundancy to check results. Specifically, the replication is implemented in the *Actorlib* [19] parallel library.

# 3. Background

The replication-based approach to detect errors and balance the program load is built in the C++ *Actorlib* library [19] on top of the parallelization library UPC++ [22].

## 3.1. UPC++

Unified Parallel C++ (UPC++) [22] is a parallel programming library built on top of C++. The library implements Partitioned Global Address Space (PGAS) in which each process in the program possesses a private and a global address space as shown in Figure 3.1. These processes are also called ranks in this context.

In order to access objects in the global address space, global pointers are used while local pointers contain addresses in the local address space. To create a pointer in the global address space in contrast to a local one, the *upcxx::global_ptr* data type can be used. The instantiation of the pointer can be done through the *upcxx::new_* or *upcxx::new_array* method. When working with a pointer in the global memory locally, the pointer can be downcasted to an ordinary C++-pointer via the *local* method of a *upcxx::global_ptr*.

In UPC++, the Single Program Multiple Data (SPMD) concept is used, meaning that every rank in the execution calls the same program with an own memory segment. For communication between ranks, UPC++ offers no implicit method in order to encourage programmers to avoid high-cost data movements. Instead, the library offers an explicit set of methods for this functionality. One important method for communication between ranks are Remote Procedure Calls (RPCs). An RPC allows the programmer to invoke a function on a remote rank with the opportunity to send local data as parameters. The result of the function can then be retrieved locally by waiting for the typed *upcxx::future*. Generally, arguments for an RPC must be serializable trivially. Alternatively, it is possible to define an own serialization. In order to not limit the programmer to write high-scale programs, all remote-memory operations including RPCs are executed asynchronously. The returned futures can be combined in various ways when for example multiple RPCs are sent at once. The most basic way is the *when_all* method waiting until all passed futures are finished.

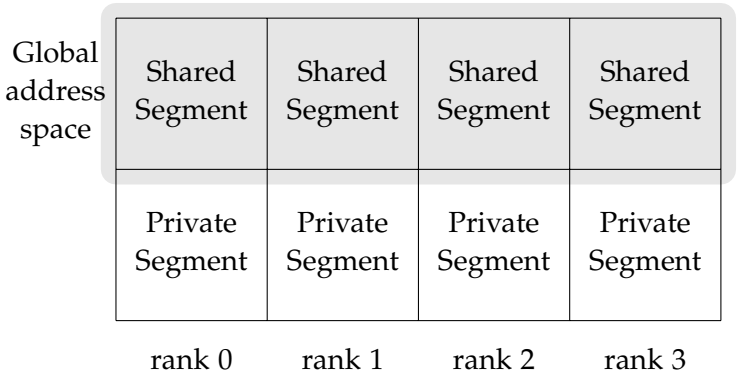



Figure 3.1.: Visualization of the PGAS used in UPC++ for a simple case using four ranks. Each rank in the execution contains a local private memory segment and a shared memory segment. During execution, a rank's pointer to its own shared segment can be downcasted to a local pointer and used as such in computation [22].

## 3.2. Actor Library

*Actorlib* is a parallel library built on top of UPC++ designed to achieve a better programmability [19]. Instead of running certain code only rank-wise, *Actorlib* offers an abstraction using actors running in an *ActorGraph*. The actors are designed to each concurrently execute a program. While one actor cannot access the data of another actor directly, channels can be established between them enabling the transfer of data. These channels carry a FIFO structure and allow for non-blocking computations by the actors even when the state of the channel changes. The actors themselves go through their states of computation independently of each other which enables them to execute in parallel while avoiding some common problems in parallelism like deadlocks.

This model originated from *Hewitt* [11] and *Agha* [1] with *Actorlib* specifically using the newer FunState actor model from actorX10 [21]. While the actorX10 library is based on the same model, *Actorlib* provides a number of improvements in terms of optimization and compatibility to common C++ libraries.

In the FunState actor model, the *ActorGraph* is a multigraph $G_a = (A, C)$ with a set of actors $A$ and channels between those actors $C$. The actors $a \in A$ are represented by a tuple $a = (ID, r, I, O, F, R)$ with an unique name $ID$, an UPC++-rank $r$ which the actor is placed on, a set of *InPorts* $I$ and *OutPorts* $O$, a set of functions $F$ and a finite state machine $R$. In order to establish the channels between actors, each *OutPort* of an

actor is connected to an *InPort*. Then, data can be written into an *OutPort*, which in turn is sent to the channel connected to the corresponding *InPort*. There, the actor to which the *InPort* belongs can read the data.

In *Actorlib*, the abstract base classes *Actor* and *ActorImpl* are defined. While this class takes over the internal UPC++ calls and some other management tasks, the user of the library implements it to fit a specific application. In particular, the programmer adds the computations each actor should contribute to the execution into the *act* method of the class. In order to add new actors or connect channels between them, the user can use the methods given by the *ActorGraph* class, which can be referred to through a *DynamicActorGraph*. Each rank has one *ActorGraph* object containing and managing the local actors.

*Actorlib* offers multiple execution strategies for the *ActorGraph*. The first one is rank-based with the actors being partitioned to the UPC++ ranks and locally executing sequentially. Then, it is possible to choose a thread-based execution where actors within a UPC++ rank are executed in parallel using C++ threads. Lastly, the task-based execution strategy is based on OpenMP tasks. Here, each time the *ActorGraph* iterates over the actors, an OpenMP task is created for each actor calling its *act* method and progressing the internal UPC++ backend. This method also allows actors within a rank to execute in parallel while, in contrast to the thread-based approach, creating as many tasks as necessary for the execution instead of an arbitrary number.

## 3.3. Silent Data Corruption

Especially in large-scale architectures, computations made by CPUs are not always correct. During simple calculations, the processor might flip a bit, invalidating the result. These so-called Silent Data Corruptions (SDCs) can have multiple sources [9].

The first one is an error in manufacturing or designing the CPU. As the processor consists of a large amount of transistors, it is probable that some of them are placed slightly off or manufactured with an error. This can lead to an inconsistent arrival time of signals, which in turn can cause bitflips. Other common mistakes in manufacturing can also cause deviances in the voltages or power thresholds of the transistors, which also can result in the device operating inconsistently. Some of those errors in the transistors can already be identified in tests during manufacturing and fixed before the CPU gets into circulation. Yet, errors in the transistors can manifest at any time even after the CPU already is in circulation. This can happen after weeks, years or at any

time after being shipped without being noticed by the manufacturer [9].

Another source of errors in the transistors is general degradation. After using a CPU, the components start getting weaker and especially with all components not being worn down equally, some failures can be produced. Still, this phenomenon is much rarer than manufacturing errors since CPUs have error correction mechanisms to protect the devices against degradation to some degree. Still, when a device is used beyond their rated life, the silicon within the transistors wears out too much for the error correction to handle and the CPU is observed to produce evermore failures. Such failures in the transistors tend to propagate from the hardware to the application and can influence an application.

Especially with an increasing amount of CPUs in a cluster, transistor failures become more common and present a considerable problem. For example in *Actorlib*, a SDC in the hardware can be passed on to the program and spread through the parallel execution. A bitflip in one actor can influence the messages sent and easily propagate to the entire *ActorGraph*, which in certain application can produce errors of high degrees.

## 3.4. Pond

The implementation of actor replication in *Actorlib* is tested through the shallow water proxy application Pond [19]. Pond is based on the code packages SWE [4] and SWE-X10 [20] which offer a similar way of simulation shallow water scenarios. This approach uses the two-dimensional shallow water equations

$$\begin{bmatrix} h \\ hu \\ hv \end{bmatrix}_t + \begin{bmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{bmatrix}_x + \begin{bmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{bmatrix}_y = S(t,x,y). \tag{3.1}$$

In this system of equations, $t$ is the time, $h$ is the height of the water, $u$ is the velocity in the $x$-direction and $v$ is the velocity in the $y$-direction. The squared brackets denote partial derivatives for the values of $t$, $x$ or $y$. In SWE, $S(t,x,y)$ is a term chosen to account for bathymetry. The simulated domain is approximated through a uniform grid, the blocks of which can be executed in parallel. In order to calculate the explicit Euler timestep $t^n \to t^{n+1}$ of a grid cell at the coordinates $(i,j)$, the Finite Volume scheme [15]

$$Q_{i,j}^{n+1} = Q_{i,j}^n - \frac{\Delta t}{\Delta x}(\mathcal{A}^+ \Delta Q_{i-\frac{1}{2},j}^n + \mathcal{A}^- \Delta Q_{i+\frac{1}{2},j}^n) - \frac{\Delta t}{\Delta y}(\mathcal{B}^+ \Delta Q_{i,j-\frac{1}{2}}^n + \mathcal{B}^- \Delta Q_{i,j+\frac{1}{2}}^n) \tag{3.2}$$

is used. In this equation, $Q^n_{i,j}$ denotes the value $[h_{i,j}, (hu)_{i,j}, (hv)_{i,j}]$ at the time $t^n$. Furthermore, $\Delta t$ is the length of the timestep and $\Delta x$ and $\Delta y$ are the grid size in the direction of the $x$- and $y$-axis. Lastly, the net updates $\mathcal{A}^{\pm}\Delta Q^n_{i\pm\frac{1}{2},j}$ and $\mathcal{B}^{\pm}\Delta Q^n_{i,j\pm\frac{1}{2}}$ denote the effect of the accumulated numerical fluxes computed with a Riemann solver from a cell's neighbors. These calculations are made within the *SWEBlock* class, which is responsible for one grid cell.

While SWE allows for parallelization of the *SWEBlocks* with MPI processes or OpenMP threads, SWE-X10 bases the parallelization on the actorX10 library. With actorX10 also being an actor-based parallelization library, this implementation is closely followed by the implementation of Pond in *Actorlib*. In the actor-based approach, each actor in the *ActorGraph* is assigned one *SWEBlock*, to which the actor delegates the computations for the simulation. In contrast to SWE itself using one rank per *SWEBlock*, the actor model allows for multiple actors, and therefore *SWEBlocks*, to exist on one rank.

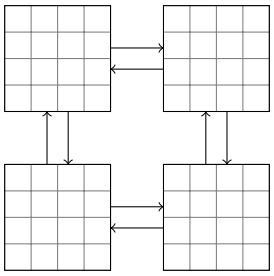


Figure 3.2.: Communication in pond with four actors. Each grid partition represents one *SWEBlock* contained in an actor. The actors communicate with their neighboring actors sending the edge data from their block.

The communication between the actors in Pond is the same as in SWE-X10 with each actor being connected to the actors with neighbouring *SWEBlocks* in the $x$-, and $y$-direction. This causes the *ActorGraph* to take the form

$$G_{pond} = (A_{pond}, C_{pond}),$$
$$A_{pond} = \{a_{i,j} | 0 \leq i < n_x \wedge 0 \leq j < n_y\},$$
$$C_{pond} = \{c_{a_{i,j},a_{i',j'}} | (i = i' \wedge |j - j'| = 1) \vee (j = j' \wedge |i - i'| = 1)\}.$$

Each actor $a_{i,j} \in A_{pond}$ then calculates fluxes until the defined end time $t_{end}$ is reached. The finite state machine of each actor takes the form shown in 3.3.

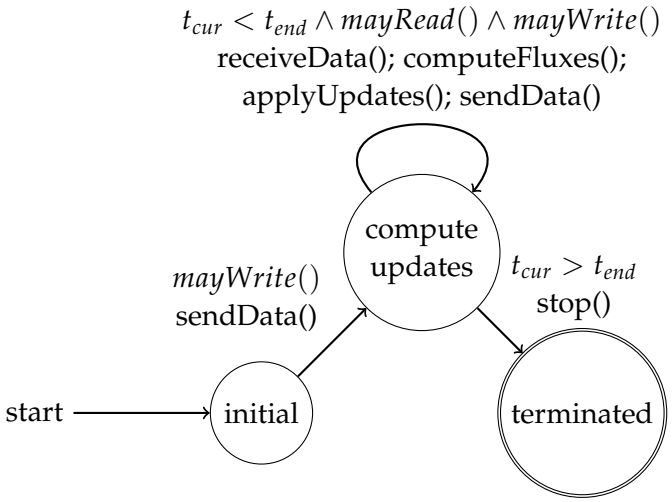



Figure 3.3.: Finite state machine of a *SimulationActor* in Pond. Here, the italicized functions are guard functions while the rest are actions. $t_{cur}$ is the current time of the simulation [19].

# 4. Replication-based Error Detection and Load Balancing

For a replication-based detection of SDCs, while there are multiple approaches, the implementation here is chosen to be built in the *Actorlib* library [20]. The replication algorithm developed here adds redundancy within the FunState actor model so that every actor in the *ActorGraph* is replicated and multiple versions of the actors are executed. This approach is similar to the one taken in *RedMPI* [10].

Both models have in common that the actual detection of errors lies within the communication between tasks or actors. In *RedMPI*, the errors are checked at the receiving task of a message. Meanwhile, in *Actorlib*, additional actors are introduced in order to manage the error checking and message passing.

For the replication in *Actorlib*, it is necessary that each actor in the *ActorGraph* produces its results deterministically. If a random element exists within an actor, a replicated actor may produce different results under the same base conditions even though no SDC occurred. Also, the type of the messages sent between the actors has to be hashable, either trivially or through a self-defined method. Additionally, this implementation also allows for the hashing of some additional data types using the *hash_reduce* method from the boost library.

## 4.1. Replication

This concept chooses to perform the replications within the FunState actor model explained in Chapter 3. Here, in an *ActorGraph* $(A, C)$ for every actor $a = (ID, r, I, O, F, R) \in A$ the replicated actor $a' = (ID', r', I, O, F, R)$ is created and added to the set of actors $A'$. The resulting set of actors $A'$ therefore can be described with

$$A' = \{a, a' | a \in A\}. \tag{4.1}$$

The changes to the connections between actors $C$ are discussed in Section 4.2.

When replicating an actor in *Actorlib*, it is important that the new actor has exactly the same starting parameters as the original actor in order to produce the same results. Still, the actors have to be able to act independently without changing the other's data remotely. In order to achieve this, it is not enough to only copy the memory from one actor to another. With most actors containing pointers, a shallow copy would enable a replicated actor to change the contents of its counterpart or even, when the actors are on separate ranks, lead to segmentation faults or undefined behavior.

With this problem in mind, it is not possible to replicate the actors in the abstract *ActorImpl* class and keep the replication completely invisible to the user of *Actorlib* as in *RedMPI*. Instead, the programmer implementing the actors has to define how to copy actors while keeping the independence between the original and the copy. For this, my implementation needs the user to implement the abstract *copy* method for the child classes of *ActorImpl*. The return value should be a deep copy with the same properties as the original actor. In *Actorlib*, a name change of the replicated actor is required as a unique identifier helping the *ActorGraph* distinguish the actors from each other, replicated or not. All actor calls are based on a map in the *ActorGraphs* on each rank mapping the unique name identifier to a *upcxx::global_ptr* which contains the corresponding actor. If now two actors had the same name, this map could not use an actor's name as its key value. To establish a new identifier for a replica returned by the *copy* method, a parameter for the name is passed and the returned actor should have that name instead of the identifier of the original actor.

While the necessary implementation of a *copy* method or the possibly needed implementation of a hash-function removes the transparency of the replication, my implementation aims to give the option of replication to the user of *Actorlib*. The user has the option to replicate any actor to a specific rank through the *DynamicActorGraph* components of their execution.

In the following applications, the *replicateAllActors* method is used to replicate all actors in the *ActorGraph* to one rank above the original actor. If an actor is on rank $i$ out of $n$ total ranks, its replica is placed on rank $i + 1 \mod n$. Through this method, the *ActorGraph* changes after replication as described in Figure 4.1.

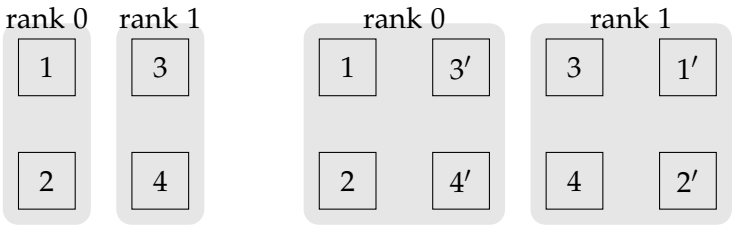



Figure 4.1.: Example with originally four actors for the placement of actors on the ranks before (left) and after (right) calling the *replicateAllActors* method.

Even though this implementation of actor replication gives some aspects of the realization of the replication to the user of *Actorlib*, it still aims to keep the replication transparent during the program execution. In order to achieve this transparency, the replicated actor has to behave exactly in the same way as the original actor. One issue here is the changed name of the replicated actor for identification purposes. Therefore, when the user asks for the name of a replicated actor, they will not receive the expected value, but instead possibly the name of a replicated actor.

In order to circumvent this problem, in the representation of the FunState actor model, the tuple of an actor is changed to $(ID, ID*, r, I, O, F, R)$ whereas $ID*$ denotes the original ID of the actor. After this change, the replication of an actor

$$a = (ID, ID*, r, I, O, F, R)$$

written as a tuple takes the form of

$$a' = (ID', ID*, r', I, O, F, R). \tag{4.2}$$

Notably, if the actor $a$ is an original actor in the program as opposed to a replicated one, it holds that

$$ID = ID*.$$

The user of *Actorlib* still has the possibility to access the actual $ID$ of an actor through the newly added *getReplicationName* method of an actor. This allows the organization of actors and direct access to some data and methods of the *ActorGraph* which require the unique identifier of an actor as a parameter. Alternatively, the original *getName* method still returns the previously expected value of $ID*$ which is the same for the original as well as replicated actors. This change particularly achieves backwards-compatibility where a user doesn't have to reprogram code based on any actors' chosen name.

When analyzing the data sent by actors in order to find errors, it is necessary to

know what type of port this message came from. In particular, it is necessary to know the data type sent in the channel as well as the size of the channel, both of which are originally not accessible information of an arbitrary *Actor* or *ActorImpl* base class after the creation of the ports. While of course the users himself has that information, this implementation aims to handle the task in the background without any further necessary action from the user. That goal is achieved through the creation of the *PortMaker* class in *Actorlib* storing the data type and the capacity of a port. For every instantiated port of an actor, an according *PortMaker* is instantiated and stored within the actor. This *PortMaker* can later be accessed to create identical ports in a replicated actor and to pass on the corresponding information.

## 4.2. Message Passing

One major challenge of replication is connecting the ports and channels in a way that does not break the communication. In the FunState actor model, the channel connected to a port is not a part of the actor that is replicated. Instead, only the *OutPorts* and *InPorts* are replicated while, when only considering the previously defined replication of the actors themselves, the channels still remain the same. If no changes were made to the communication in this model, one channel would be connected to the corresponding port of the original actor as well as the same port of the replicated actor.

As mentioned in Section 3.2, a channel can be seen as a queue collecting the messages written into the connected *OutPort*. Then, elements of the queue can be read in an *InPort* whereas that element is removed from the queue. Now, if one channel is connected to multiple *InPorts*, only the first actor to read actually receives the message while the latter port does not find the now deleted message. Due to this phenomenon, the original actor and the replica no longer receive the same input and therefore do not compute the expected results. A similar problem occurs in the outgoing connections of a replicated actor. The *OutPorts* of the actors are all connected to the same channel and with both actors producing results, they are all written twice.

If every actor in the graph is replicated for the same amount of times, the channels of the actors and their replicas could be connected in a way that both problems do not occur. While this approach was chosen in [10], in this model, such a combination is not deterministic with the original actor and the replica running on different ranks. Due to the different speeds of the ranks, it is not guaranteed that the messages arrive in a specific order and, without an additional communication protocol, it is impossible to distinguish what messages arrived from which actor. For example, if the rank of

the original actor is faster than the one of the replica, it is possible that the original actor has already written multiple messages while the replicated actor is still stuck on a previous one. Now, the receiver has no way of concluding whether all of those messages came from one actor or if the possible discrepancy between the messages comes from a SDC.

While message passing similar to *RedMPI* [10] might be a possible solution to this problem, here it was chosen to remain within the framework of the actor model and delegate the management of messages to new actors added to the *ActorGraph*. These actors are specifically created to multiplex multiple channels to one and demultiplex one channel to multiple receivers.

These actors are specifically the *MulAct* combining multiple incoming message to one output channel and the *DemulAct* distributing every incoming message to the connected *InPorts*. The *MulActs* and *DemulActs* are created whenever an actor $a = (ID, ID*, r, I, O, F, R)$ is replicated with the replica $a' = (ID', ID*, r', I, O, F, R)$ as in Equation 4.2. The resulting *DemulAct* is defined by

$$\begin{aligned} a_{demul} &= (ID_{demul}, ID*_{demul}, r', I, O_{demul}, F_{demul}, R_{demul}), \\ O_{demul} &= \{o_0^i, o_1^i | i \in I\}, \\ o_k^i &= (ID_k^i, t^i, n^i) \quad k \in \{0, 1\}, \end{aligned} \tag{4.3}$$

with each *OutPort* having the type $t^i$ and capacity $n^i$ of a corresponding *InPort* $i \in I$ in the replicated actor. The identifier of an *OutPort* $ID_k^i$ as well as the identifier of the actor $ID_{demul}$ and $ID*_{demul}$ are arbitrarily chosen to be based on the names of the original actor and its *InPorts* in this implementation.

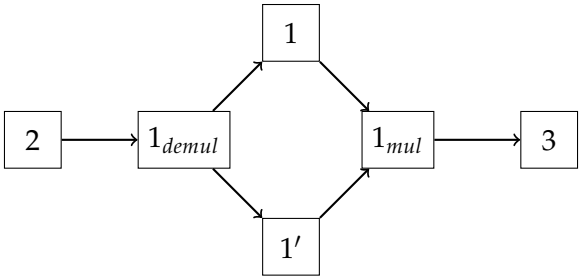



Figure 4.2.: Functionality of a *MulAct* and a *DemulAct* of an actor 1 and its replica $1'$. Before replication, the original actor 1 had an incoming channel from 2 and an outgoing connection to 3

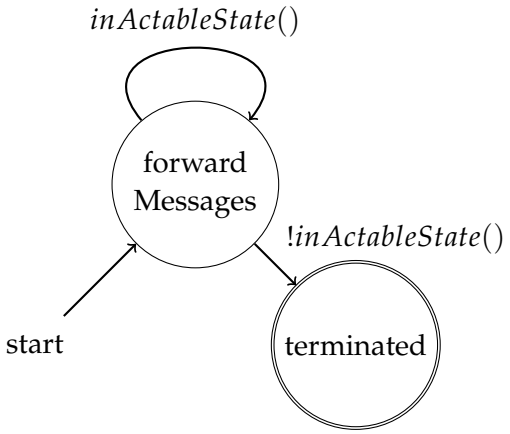



Figure 4.3.: A finite state machine for the *MulAct* or *DemulAct*. The *inActableState* is checking certain neighbours whether they are still running.

*MulActs*, in their creation, are similar to *DemulActs* in that they produce corresponding ports to the original actor. While the *DemulActs* catch messages before they arrive at an actor and its replica, the *MulActs* handle messages sent from the actor. Therefore, the *InPorts* of these actors are created based on the *OutPorts O* of the corresponding actor.

$$\begin{aligned} a_{mul} &= (ID_{mul}, ID*_{mul}, r', I_{mul}, O, F_{mul}, R_{mul}), \\ I_{mul} &= \{i_0^o, i_1^o | o \in O\}, \\ i_k^o &= (ID_k^o, t^o, n^o) \quad k \in \{0,1\}. \end{aligned} \tag{4.4}$$

Both actors are based on the same general finite state machine only with a different implementation of the used functions.

The *inActableState* function checks if any work still needs to be done by the *MulAct* or *DemulAct* and, if not, the actors are stopped. For the *MulAct*, all actors connected to its *InPorts* are checked whether they are done. As soon as they are, all messages that are supposed to be filtered and sent on have arrived at the *MulAct* and the actor does not have anymore tasks after sending these last messages on.

Here, it is notable that first the actors on the same rank as the *MulAct* are checked for their state. After, the search for running actors is expanded to all ranks. While this is the cause for some redundant code, during most of the execution only the local search will be done due to the fact that the replicated actor is put on the same rank as the *MulAct* and *DemulAct*. Therefore, as long as the replica of the original actor is running, the search for active neighbors is not expanded. This is usually the case until

shortly before the end of a run.

The double-staged search for connected running actors is added to the algorithm due to the high cost of communication between ranks in *Actorlib*. If locally no connected running actor is found, RPCs have to be sent to connected remote actors requesting their status. This is very time-consuming and should be avoided, especially in code running as frequently as the multiplexing and demultiplexing. Meanwhile, the local checks are just simple local memory acccesses. For this reason, the *DemulAct* is also chosen to not check its incoming connections for active neighbours, but instead the outgoing ones. A *DemulAct* is always guaranteed to have an outgoing connection to the replica of the original actor, which in this implementation is always on the same rank as the *DemulAct*. In contrast, the incoming connections can come from any rank the user specifies, possibly causing very large communication costs.

All in all, the replication of an *ActorGraph* $G = (A, C)$ results in a new *ActorGraph* $G' = (A', C')$. In this graph the actors $A'$ consists of

$$A' = \{a, a', a_{mul}, a_{demul} | a \in A\}, \tag{4.5}$$

with $a'$ being the replica of actor $a$ according to Equation 4.2, $a_{mul}$ being the corresponding *MulAct* as in Equation 4.4 and $a_{demul}$ as the *DemulAct* of actor $a$ following Equation 4.3. In a simple example with only two actors and the *ActorGraph* not being a multigraph the graph changes as depicted in Figure 4.4.

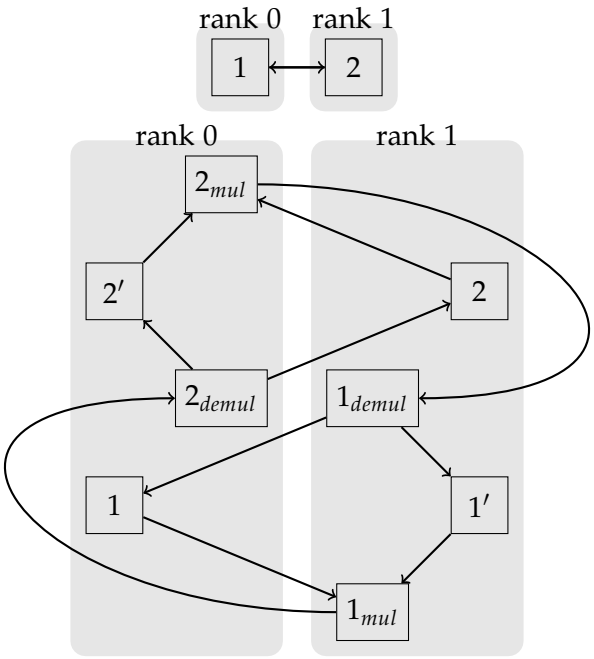



Figure 4.4.: On top is the original *ActorGraph* containing the actors 1 and 2. Here, actor 1 and actor 2 each have one *InPort* and one *OutPort* through which they communicate. The bottom image contains the corresponding *ActorGraph* after replication.

Notably, in this example, every actor only has one connected *InPort* and one *OutPort* which is the reason only one connection can exist between the two actors.

In order to define the possibility of multiple edges between two actors let

$$C \subseteq \{(s, r, o_s, i_r) | s, r \in A, o_s \in O_s, i_r \in I_r\} \tag{4.6}$$

be an extended definition of the channels within the *ActorGraph*. Here, $s$ is the sending actor and $r$ is the receiver while $o_s$ is an *OutPort* of $s$ connected to an *InPort* $i_r$ of $r$. When connecting the *MulActs* and *DemulActs*, the original connections have to be removed and afterwards, new connections have to be initialized between the ports. The resulting channels $C'$ are defined as follows:

$$\begin{aligned} C' = \{&(s, s_{mul}, o_s, i_0^{o_s}), (s', s_{mul}, o_s, i_1^{o_s}), (s_{mul}, r_{demul}, o_s, i_r) | (s, r, o_s, i_r) \in C\}, \\ &\cup \{(r_{demul}, r, o_0^{i_r}, i_r), (r_{demul}, r', o_0^{i_r}, i_r) | (s, r, o_s, i_r) \in C\}. \end{aligned} \tag{4.7}$$

In this definition, $i_0^{o_s}$ and $i_1^{o_s}$ are *InPorts* of the *MulAct* $s_{mul}$ and $o_0^{i_r}$ and $o_1^{i_r}$ are *OutPorts* of the *DemulAct* $r_{demul}$ as described in Equations 4.4 and 4.3. Using this representation,

$(s, s_{mul}, o_s, i_0^{o_s})$ and $(s', s_{mul}, o_s, i_1^{o_s})$ are the channels through which the actors send their messages to the *MulAct*. Then, the *MulAct* forwards the message to the *DemulAct* of the receiver through the channel $(s_{mul}, r_{demul}, o_s, i_r)$. Lastly, the *DemulAct* of the actor $r$ makes sure the message arrives at both the receiver and its replica through the channels $(r_{demul}, r, o_0^{i_r}, i_r)$ and $(r_{demul}, r', o_0^{i_r}, i_r)$.

In order to implement the task of receiving and sending all messages correctly in *Actorlib*, the *MulActs* and *DemulActs* delegate the forwarding of messages to so-called *PortConnectors*. This object contains three ports. In the case of a *MulAct* it has two *InPorts* and one *OutPort* while the connectors of the *DemulActs* have one *InPort* and two *OutPorts* each. For every *OutPort* in the original actor, a connector is created in the *MulAct* while every *InPort* translates to a connector in the *DemulAct*. These *PortConnectors* hold the type and the capacity of the channels connected to the *MulAct* or *DemulAct*.

The creation of these *PortConnectors* is delegated to the previously introduced *PortMaker* objects. These objects are initialized whenever a port is created in an actor. Then, when that actor is replicated, each *PortMaker* creates the corresponding ports for the *MulAct* and *DemulAct* encapsuled in *PortConnector* objects. This is done through the *createInPort* method which takes an actor as an argument and returns the *PortConnector* that is after added to the actor.

The algorithm for replicating an actor in *Actorlib* then looks as depicted in Algorithm 1.

**Algorithm 1:** Algorithm for replicating an actor $a$

```
   Data: original actor a, ActorGraph ag
 1 a' ← a.copy()
 2 ag.addActor(a')
 3 if a'.hasOutPorts() then
 4 |   a_mul ← MulAct()
 5 |   ag.addActor(a_mul)
 6 |   foreach maker ∈ a'.OutPortMakers do
 7 |   |   connector ← maker.createOutPort(a_mul)
 8 |   |   a_mul.addPortConnector(connector)
 9 |   end
10 end
11 if a'.hasInPorts() then
12 |   a_demul ← DemulAct()
13 |   ag.addActor(a_demul)
14 |   foreach maker ∈ a'.InPortMakers do
15 |   |   connector ← maker.createInPort(a_demul)
16 |   |   a_demul.addPortConnector(connector)
17 |   end
18 |   connections ← ag.getConnections(a)
19 |   ag.removeConnections(a)
20 |   ag.reconnectChannels(connections)
21 end
```

While this is the general algorithm for replication, there are some aspects to it that are not shown here for simplification. Most importantly, when replicating an actor to another UPC++ rank, the actors $a'$, $a_{mul}$ and $a_{demul}$ have to be available in the local memory of the rank to which they are added. Specifically in this implementation, the actor $a'$ is initialized with a global pointer on the original rank which makes it possible to send it to the new rank and access it locally there. In contrast, in the case of the *DemulAct* and *MulAct*, the procedure of creating them and filling their *PortConnectors* is completely delegated to the new rank via an RPC. There, the previously locally added replica of the original actor can be used to access the *PortMakers* and initialize $a_{mul}$ and $a_{demul}$.

Another aspect of the replication not mentioned here is the replica's, the *MulAct*'s and the *DemulAct*'s choice of identifier. When calling the *copy* method on an actor a unique name has to be passed as an argument that the *ActorGraph* then uses to identify the replica. Similarly, when calling the constructor for the *DemulAct* or *MulAct*, a name

has to be chosen as well.

Lastly, when creating the new connections in the *ActorGraph*, first the new channels are determined using the previous connections of actor *a*. Then, all connections of the original actor are removed before the new ones are reinitialized.

## 4.3. Error Detection

The first use of replicated actors is the checking of sent messages for SDCs. In order to do this, the messages of both the original and the replica actor have to be compared and checked whether they are equal or if there are possible bitflips.

In *RedMPI*, it is proposed to check both MPI tasks' messages by sending a hash to the replica or the original receiving MPI task. Then, the receiver calculates the hash of the message coming from one of the two sending actors and compares it to the hash sent by the other.

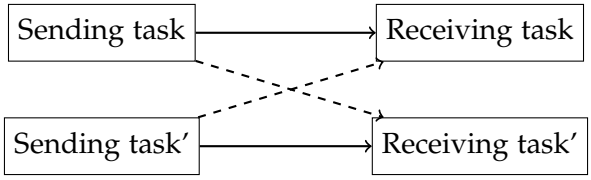



Figure 4.5.: Communication in the *RedMPI* MsgPlusHsh method. The sender and its replica send messages to the corresponding receiver while sending a hash of the message along the dashed arrows [10].

In the actor model, due to the structure of the message forwarding system, this way of cross-referencing is changed. When an actor and its replica both send a message, they both send it to their *MulAct* before it arrives at the actual receiver. Since, in this work, we assume that no errors happen within the channels, the actors sending an additional hash of their message is an unnecessary overhead. Instead, the *MulAct* receiving the messages from the original actor and the replica computes the hash of all received messages and compares those values. Meanwhile, the actors only send their message itself and completely leave the task of error checking to the *MulAct*. This method effectively achieves the same result as the implementation from *RedMPI*.

When checking messages in the *Actorlib* implementation, each *MulAct* has to check all incoming port connections for errors. Therefore, the ports and message passing of the

*MulActs* are managed in the corresponding *PortConnectors* of the actor, specifically in this case the sub-class of *PortMultiplexers*. These objects are assigned to one port of the original actor each. The connector has two *InPorts*, one connected to the original actor and the other connected to the replica. When receiving messages on these ports, the *PortMultiplexer* checks them and sends them to the receiver through its *OutPort*. Each *PortMultiplexer* has two queues in which it stores the hash values of the incoming messages before they can be checked. Then, every time a message is received in any of the two *InPorts*, the following algorithm is executed to detect SDCs:

**Algorithm 2:** Algorithm for checking for a SDC in a *PortMultiplexer*

**Result:** Was a SDC detected?
**Data:** Queues $q_0$ and $q_1$ for the two *InPorts* of the *PortMultiplexer*

```
1 if q0.size()> 0 ∧ q1.size()> 0 then
2 |   return q0.pop() ≠ q1.pop()
3 end
4 return false
```

Here, it is notable that it is never necessary to iterate over the queue of stored hash values since only one element can be checked at a time here. With Algorithm 2 being executed each time a message is received, at least one of the two queues contains only one or no elements. This occurs due to the fact that if both queues contain an element, the oldest element is checked for errors and then removed.

In this implementation, once a SDC is detected, the program cannot be continued and has to be terminated or restarted due to the double replication not being able to correct detected errors. This method of using replication could instead be combined with a checkpointing system which allows the program to rollback to a previous checkpoint once an error is detected. Still, this implementation only focuses on the detection of errors while leaving their handling to further work.

## 4.4. Load Balancing

Replication may be attractive for load balancing, as an imbalance created by a delayed rank can be made up for by its replica. Here, it is expected that with both the replicated actor as well as the original producing results, the application can use whichever result arrives faster and therefore reduce wait times. In order to do this, there first needs to be a change to the message passing.

In a general pessimistic error detection algorithm, a *MulAct* has to wait for mes-

sages coming from both an original actor and its replica. With this technique, it is made sure that the error will not spread. On the other hand, waiting for both messages can be time-consuming, especially in unbalanced scenarios due to the original and the replica actor being on separate ranks. If, for example, the rank of the replica and of the receiver are much faster than the one of the original actor, the receiver might have to wait for a long time until the original actor reaches the right point in execution.

When forwarding messages immediately, it is important to only send the messages of the actor that is further in execution. Since the previous implementation of replication in *Actorlib* already features a queue to store the hash of messages previously received from one of the actors, the *PortMultiplexers* only have to be changed to immediately forward the messages when receiving it. Now, in order to not send the messages twice, the *PortMultiplexer* uses the following algorithm.

**Algorithm 3:** forwardMessages

```
   Data: InPorts in_0, in_1, OutPort out
 1 while in_0.available() do
 2 |   value ← in_0.read()
 3 |   value_hash ← hash(value)
 4 |   q_0.push(value_hash)
 5 |   if q_0.size() > q_1.size() then
 6 |   |   out.write(value)
 7 |   end
 8 end
 9 while in_1.available() do
10 |   value ← in_1.read()
11 |   value_hash ← hash(value)
12 |   q_1.push(value_hash)
13 |   if q_1.size() > q_0.size() then
14 |   |   out.write(value)
15 |   end
16 end
```

Here, it is still assumed that actors are deterministic and the messages from replicas and originals arrive in the same order. Therefore, when looking at the queue of hashes from messages, the actor with the longer queue necessarily has sent more messages in its execution and therefore its messages should be forwarded while the other actor's messages loose their significance for the matter of load balancing.

In the case of both actors having same sized queues, which in this implementation means both queues have the size zero, both the replica and the original are at the same point of execution in terms of messages sent and any newly arriving message can immediately be forwarded. As long as the other actor is not ahead of an actor, the message has not been previously sent and a received messages can just be forwarded in the corresponding *OutPort*.

The use of replication as a method of load balancing is expected to have the largest effect in scenarios where the ranks are very unbalanced in terms of speed. When using redundancy for load balancing, an actor and its replica are put on separate ranks, one of which is likely to achieve a relatively fast runtime. Generally, in these scenarios, there are other possible approaches like migrating actors from a slow rank to a faster one, but most methods suffer from having to predict which rank will be fast or slow in the following execution in order to find a dynamic solution. Using replication, it is not necessary to make any prediction on the future of the run but instead the faster solution is always automatically chosen. On the other hand, a double redundancy initially induces an overhead factor of at least two due to the double execution of each actor with an additional overhead of the *MulActs* and *DemulActs*. While there might be scenarios where the overhead of waiting for other actors surpass the overhead of replication, the initial overhead of redundancy in itself definitely is a considerable factor.

| *MulAct* | *DemulAct* | *PortConnector* |
|---|---|---|
| This actor manages all messages coming from an original actor and its replica. | This actor distributes all incoming messages to an original actor and its replica. | The *PortConnector* is an abstract class offering the $forwardMessages$ method. This base class is used for multiplexing and demultiplexing. |
| *PortMultiplexer* | *PortDemultiplexer* | *PortMaker* |
| This is an implementation of the *PortConnector* class containing one *OutPort* and two *InPorts* connected to an actor and its replica. A *MulAct* contains one *PortMultiplexer* for each *OutPort* of the original actor. Here, the error checking and load balancing is performed. | The *PortDemultiplexer* is the second implementation of the *PortConnector* class containing one *InPort* and two *OutPorts* connected to an actor and its replica. A *DemulAct* contains one *PortDemultiplexer* for each *InPort* of the original actor. | This class is responsible for creating *PortConnectors*. A *PortMaker* is instantiated whenever an actor creates a port. That object offers the necessary methods to create a *PortMultiplexer* or a *PortDemultiplexer* for the port with which it was created. |

Table 4.1.: Overview of added classes to *Actorlib* involved in the replication.

# 5. Results

The implementation of replication in *Actorlib* was tested on the CoolMUC-2 cluster of the Leibniz Supercomputing Centre (LRZ). The CoolMUC-2 cluster consists of 812 28-way Haswell-based nodes with FDR14 Infiniband interconnect [14]. The tests here are performed with up to 16 nodes with each node using all available 28 cores.

The benchmarks of this extension to *Actorlib* were all performed in the application Pond as described in Section 3.4. This implementation of Pond uses the replication of every actor in the simulation. This is done before the start of the run by the *ActorGraph* on rank 0 using the method *replicateAllActors*. The baseline for the tests is the original pond application without any added redundancy in the computations in the form of double replication of actors.

Firstly, the performance of the Pond application is tested with grid sizes of $8000 \times 8000$ and $16000 \times 16000$. These tests are performed with a patch size of $250 \times 250$ and a simulation end time of 4. The scenario chosen for all tests using pond is the *PoolDrop* scenario. The node count for the application is set between 2 and 16 nodes with runs using 4 and 8 nodes in between. Here, the implementation is tested with the serial execution of the *ActorGraph* without multi-threading using OpenMP or C++ threads within the ranks. In this test case, each scenario is only executed once. Due to the large runtimes of these runs, it is expected that system noice averages out so single measurements can accurately represent the runtime differences between the baseline and the tested implementation.

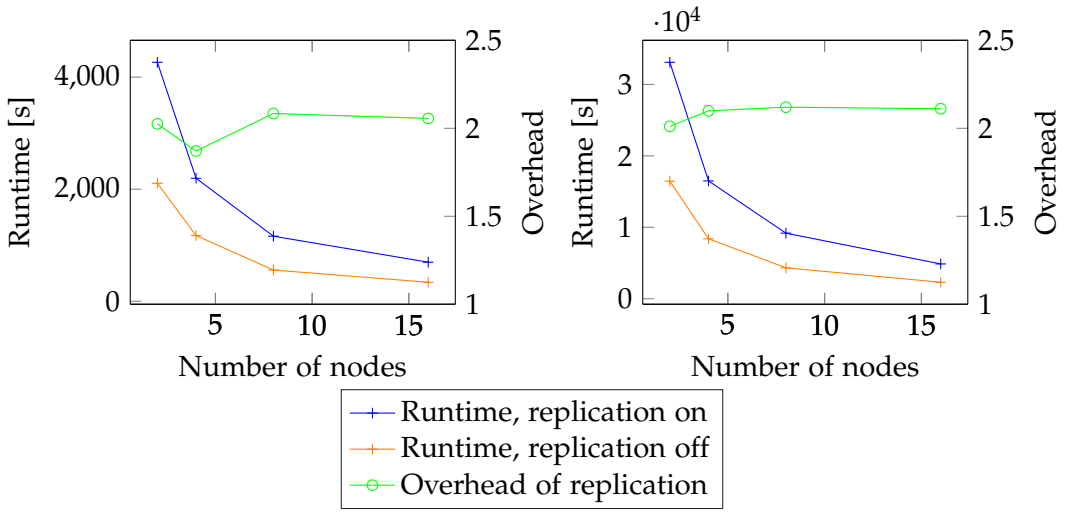



Figure 5.1.: Results of runs of Pond with a grid size of 8000 × 8000 (left) and 16000 × 16000 (right) with 2-16 nodes. The end time of the simulation is set to 4 with a patchsize of 250 × 250. The orange plot represents the runs without replication while the blue plot shows the runtimes with replication. The green plot represents the overhead induced by replication.

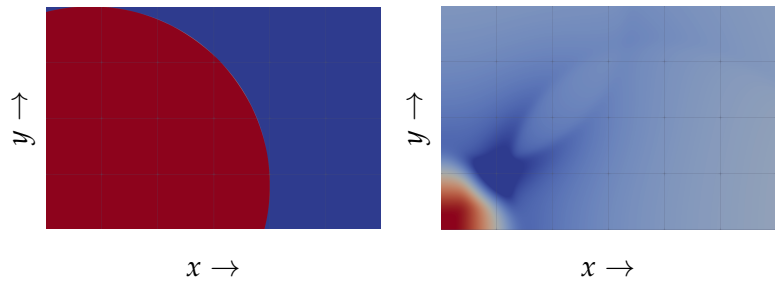



Figure 5.2.: *PoolDrop* scenario of Pond at the start of the simulation (left) and at the end of the simulation (right). The results are visualized in ParaView with red and blue values representing high or low heights at certain coordinates.

## 5.1. Overhead of Replication

In Figure 5.1, the plots in orange and blue represent the runs of the baseline and the execution with replication respectively. The corresponding scale for the runtime of the executions is shown at the left of each plot measured in seconds.

Meanwhile, the green lines represent the overhead incurred by the replication determined through the formula

$$OH = \frac{time_{replication}}{time_{baseline}}. \tag{5.1}$$

Here, $time_{replication}$ and $time_{baseline}$ denote the measured runtime using replication and in the baseline. The scale for this overhead is on the right side of each plot.

In the plots from Figure 5.1, it is already possible to qualitatively see the scaling of the replication. When increasing the node count for the runs, the implementation with replication evidently scales similarly to the baseline executions in orange. Both in the smaller and the larger grid size the overhead incurred by the replication remains stable on a level slightly above a factor of 2.

With no larger imbalances occurring in these scenarios, the load balancing is not able to achieve a large speedup. For this reason, the minimum expected overhead incurred by redundant computations of replicated actors here is expected to be around a factor of 2 only from the replicated actors. The measured data containing the management of the messages between actors achieves a value only slightly above that threshold.

In order to further investigate the overhead incurred by this form of replication, the runtimes and overhead when scaling the computational resources to the redundancy of replication are shown in Tables 5.1 and 5.2. In this case of double replication, the node count is doubled while still processing a problem of the same size as the baseline. Here, the runs without replication using 4 and 8 nodes are compared to the runs with replication using 8 and 16 nodes and 2 and 4 nodes.

Table 5.1.: Runtime and overhead comparison in Pond runs with a grid size of 16000 $\times$ 16000, a patch size of 250 $\times$ 250 and a simulation time of 4. Here, the runtimes of pond without replication using 4 and 8 nodes are compared to the data with replication using 8 and 16 nodes.

| Nodes | Original runtime [s] | Replication runtime [s] | Overhead |
|---|---|---|---|
| 2-4 | 16455 | 16491 | 2.19% |
| 4-8 | 8390 | 9180 | 9.41% |
| 8-16 | 4330 | 4870 | 12.47% |

Table 5.2.: Runtime and overhead comparison in Pond runs with a grid size of 8000 × 8000, a patch size of 250 × 250 and a simulation time of 4. Again, the amount of nodes in the runs with replication is scaled to double the amount from runs without replication.

| Nodes | Original runtime [s] | Replication runtime [s] | Overhead |
|---|---|---|---|
| 2-4 | 2106 | 2195 | 4.02% |
| 4-8 | 1173 | 1161 | -1.02% |
| 8-16 | 557 | 697 | 25.13% |

The overhead induced by replication shown in Tables 5.1 and 5.2 mainly comes from the multiplexing and demultiplexing of the messages. While the checking and forwarding of the messages was observed to be very performant, finding active neighbours creates a significant overhead. While the algorithm is already improved by the two-phase checking first looking for active neighbors locally, only these computations still make up the majority of the computational time in the *MulActs* and *DemulActs*. In order to find connected neighbours in the *ActorGraph*, the *PortGraph* storing all connections has to look up the actor in question from the connections in the *ActorGraph*. After receiving the names of the connected actors, it is possible to look up through the *ActorGraph* whether or not they are active. These lookups in the *PortGraph* and the *ActorGraph* have proven to create a large overhead.

It might be possible to only check the connected neighbors occasionally instead of in every *act* call of the *MulActs* and *DemulActs*. This could be done by checking whether or not messages were received while assuming that the neighbours are guaranteed to be active when they send a message to their *MulAct* or *DemulAct*. Yet, this approach is based on the messages arriving in regular intervals. Due to the inability to predict when messages can be received in the future, this approach might even increase the overhead. Especially in the case of Pond, only very irregularly messages are received by the *MulActs* and *DemulActs* which makes such an approach to reducing the overhead infeasible.

Alternatively, pointers to the neighboring actors of a *MulAct* or *DemulAct* could be stored directly circumventing the lookups in the *PortGraph* and *ActorGraph*. Instead, checking the neighbours could be done only through efficient memory accesses. The downside of this approach is that it in itself proposes a static structure of the *ActorGraph*. Therefore, when the structure of the *ActorGraph* or the *PortGraph* changes, these changes have to be broadcasted to all *MulActs* and *DemulActs* so they can refresh their pointers

to contain their possibly new neighbours.

## 5.2. Error Injection

### 5.2.1. Error Injection Framework

In order to check the functionality of the error detection algorithm within the replication, the occurrence of SDCs is simulated. Due to the low chance of SDCs naturally happening, bitflips are artificially inserted into Pond. This method is favorable to finding natural SDCs for testing in order to achieve meaningful results that are also reproducible.

Similarly to the testing method in *RedMPI*, ten experiments are done in each of which SDCs can be inserted. While *RedMPI* inserts the errors at a rate of $1 : 5,000,000$, this implementation chooses to randomly insert these errors with a probability of $1 : 5,000,000$ in order to achieve different, random results in the same scenario. This probability for an artificial SDC is evaluated each time the water heights of a block are calculated. Then, if an error should be inserted, the water height at a certain point in the corresponding block has its last bit flipped. While the choice of what bit of the height is flipped is in this case arbitrary, it can only distort the results of the experiment if it regularly causes multiple hash values of the height values to collide. Using the *hash_combine* method from the library boost to calculate the hash of a sent vector of floats/doubles, this is observed to not be the case.

In this implementation, I chose two different spatial positions in the grid in which an error may be inserted. One of these positions is the edge of a *SWEBlock*. In this case, an error is added to the heights of the left and the right side of a block. This causes the error to immediately spread to the bordering blocks, in the communication to which the error detection is expected to catch the artificial SDC.

The detection of a SDC might be significantly more difficult to detect if the corruption occurs in the middle of a block. In this case, the error first has to spread to the edge of a block and only then it can be detected by the error checking algorithm during the communication. If the simulated time is too short for the error to spread, the error cannot be found with this method. Additionally, an error might even out throughout the simulation time.

Another interesting aspect of the error injection is to evaluate how an error might affect the program. In order to check the effect of the injected errors without detecting

them, 100 additional tests are introduced to be run with and without actor replication. In these cases, a smaller grid size of 2000x2000 is chosen for pond with an endtime of 4. Due to the fewer computations these runs perform, the chance of an error occurring is increased from the previous value of $1 : 5,000,000$ to $1 : 350,000$. This number is chosen so enough runs perform an error injection while other runs do not. The latter runs are used to determine whether or not false positives occur.

The errors are again possibly inserted whenever the heights of the grid in a *SWE-Block* are updated. Then, if a SDC is to be inserted, the height in the middle of the block is chosen and a random bit of the float is flipped. With this approach the error created can randomly be either small or large relative to the original height.

SDCs are typically dangerous if, instead of causing a program failure, they remain undetected and distort the results of computations. Especially in environments without much error tolerance, one bitflip might cause drastically different results than expected. Alternatively, there are applications that require a large degree of accuracy which can possibly not be achieved when an error in the execution is not detected. For this reason, it is of interest whether the injected bitflips can be detected through a program failure or if they would go unnoticed without using error detection.

### 5.2.2. Error Detection

The effect of a bitflip can vary immensely depending on how fault-tolerant the program is in which it happens and what value is changed. In the case of Pond, the bitflips of height values at the middle of a block created errors that were observed to spread to the edges of the block and to the blocks of other actors. Yet, when only flipping a bit in one height value, the errors tend to become small to a degree that they can't visually be identified in the resulting simulation. In order to see what results more errors can have on pond visually, the error insertion chance was first set to $1 : 200$ while always switching the bit for the height's sign in case of an error.

In this case, as seen in Figure 5.3, some artifacts are visible but pond still remains relatively stable. The artifacts created by error insertion start being visible only in certain timesteps with an insertion chance of $1 : 20,000$ as shown in Figure 5.4.

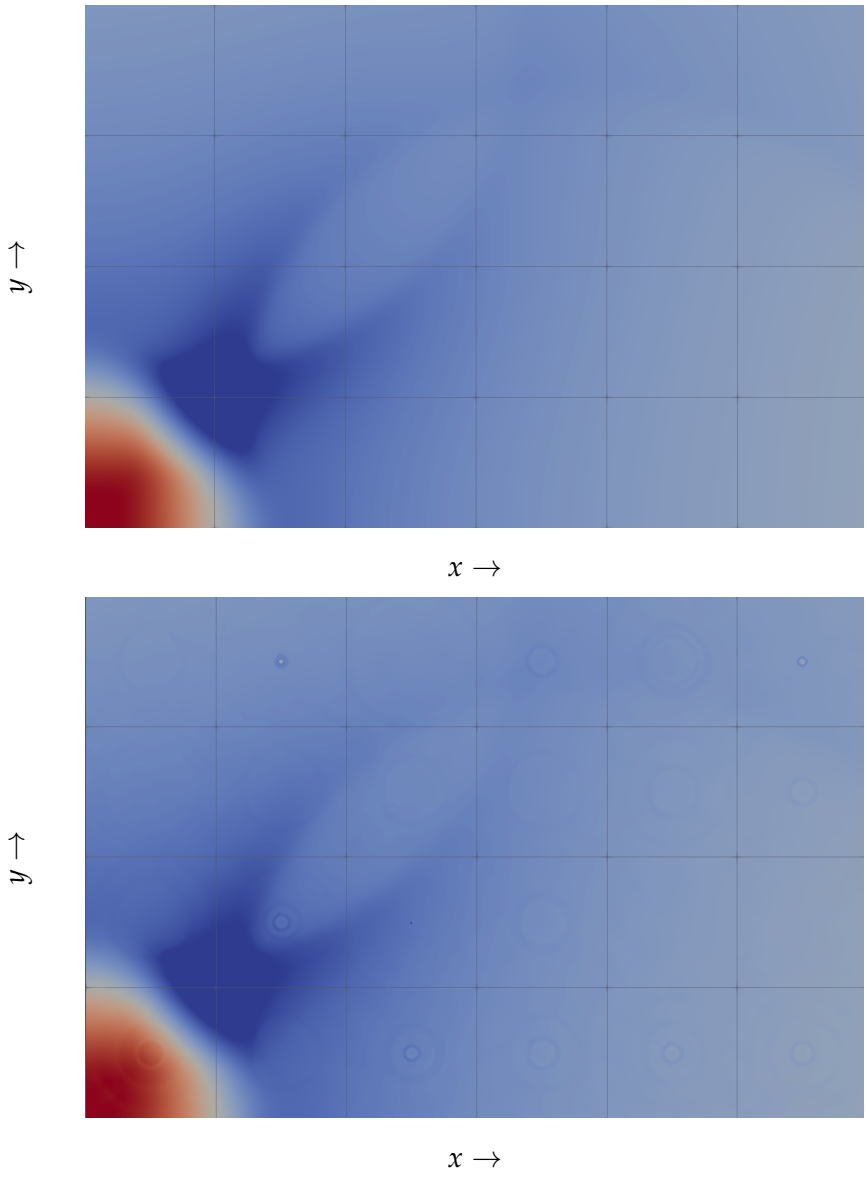



Figure 5.3.: Part of the end step of a pond simulation visualized through ParaView. In the image above, no errors were inserted. Below, errors were inserted with a chance of 1 : 200 creating some artefacts. These errors took the form of a bitflip at the sign of the middle height of a block.

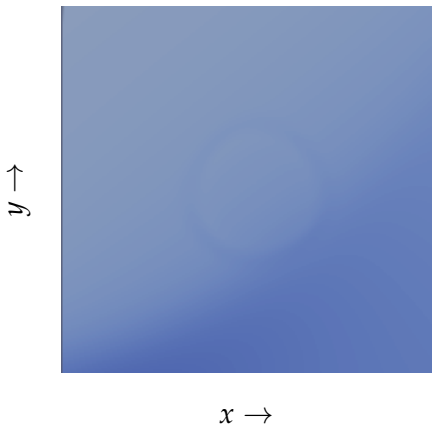



Figure 5.4.: A part of a middle time step of a pond simulation visualized in ParaView. Here, with a chance of $1 : 20,000$, the sign bit of the middle height of a block was flipped. In this case, only certain timesteps in the simulation showed small artifacts that evened out throughout the simulation time.

Pond was observed to produce very stable results even when, instead of a simple bitflip, a larger error was inserted to a height. In Figure 5.5, when an SDC should be inserted, instead of flipping one bit, the height in the middle of a block was set to the maximum float value defined by C++. While this error was possible to spot in the produced fluid simulation, the results remained stable and converged to results similar to the baseline without an error.

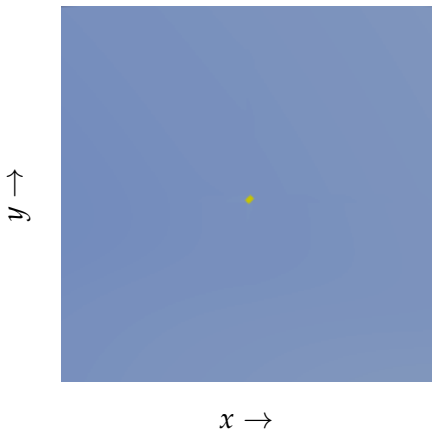



Figure 5.5.: This is a visualization of one block of pond at the end time of the simulation from ParaView. In this case, with a chance of $1 : 200,000$, the middle height value of a block is set to the numeric maximum float value.

In order to find out how well errors can be detected through replication, ten runs are tested for each an inserted error in the middle and at the edge of a block with the last bit of the water height being flipped. Here, both methods to insert a corruption use the same random seeds in order to create the same framework condition for the tests. During ten experiments each with a chance of $1 : 5,000,000$ for an error to be inserted, eight runs had an error while the other two runs did not have an artificial SDC.

When inserting the error at the edge of a block or when corrupting the middle of a block all inserted errors were detected by the error detection algorithms. Additionally, in the experiments without data corruptions, this implementation produced no false positives detecting a data corruption where there was none.

The second set of tests performed inserts all errors in the middle of a block with 100 cases but a smaller grid size. When performing these experiments, 47 runs had an error injected while 53 runs did not have an artificial bitflip. Out of the runs with an error injection, none were observed to fail. Instead, all executions terminated as if no error had occurred. Meanwhile, the tests were performed using the same random seed while enabling actor replication with error checking. Here, all but four errors were still correctly detected while again no false negatives occurred. The errors that were not detected were inserted into the middle of a block too late in the simulation for them to spread to the edges where the error detection is located.

## 5.3. Replication for Mitigating Dynamic Load Imbalances

The load balancing through replication is also tested in the pond application with a grid size of $8000 \times 8000$ with 8 nodes on the CoolMUC-2 cluster. While other methods for load balancing achieve good results for static imbalances, the approach through replication aims to balance dynamic imbalances which other approaches might not be able to predict.

Here, these dynamic imbalances are inserted into the pond library through slowing down half the ranks for a certain time. After the slowed down ranks complete their slowdown time, they are allowed to continue their execution and the other ranks are waiting. In this implementation, at any time, every odd rank or every even rank is allowed to execute while the others wait. This approach is expected to achieve good results in combination with the approach of replication used in pond. There, every replica of an actor on a rank $n$ is put on rank $n + 1$. Then, with every second rank being slowed down, every actor can either work itself or it has to wait and the replica on the other rank can take over.

When testing this application with a serial execution of actors, there is a problem when placing the *MulActs* and *DemulActs*. In pond, these actors are placed on the same rank as the corresponding replica actor. If, now, the rank of the replicated actor is slowed down and its actors cannot execute, the *MulActs* cannot act either. When this happens, the messages of the running original actor arrive at the *MulAct* to be sent on and checked for errors but the message will not be forwarded due to the slowdown. Therefore, the load balancing of the replication does not take effect. Another difficulty of creating imbalances in *Actorlib* is the size of the channels between actors. When an actor is slowed down, the replicated actor takes over computing results and sending messages. Now, when a receiver of those messages is suspended, it cannot read anymore messages and the channel can fill up. In particular with using the serial execution of the *ActorGraph*.

In order to avoid these two problems, the channel size of the actors is increased from 128 to 4096. Also, the slowdown to the actors is applied by the *ActorGraph*. Therefore, when the rank of an *ActorGraph* component is slowed down, the actors in the simulation are skipped while the *MulActs* and *DemulActs* are still executed. The results of this method using the same measure as Tables 5.1 and 5.2 can be seen in Table 5.3.

Alternatively, the OpenMP implementation of the *ActorGraph* is used in order to

Table 5.3.: Runs of imbalanced Pond in a serial execution with an increased channel size of 4096. Here, the increased channel size in combination with the imbalances require a large amount of memory during execution. For this reason, these tests were not performed with two nodes. The run uses a grid size of $8000 \times 8000$ with a patch size of $250 \times 250$ and an endtime of 4. The runtime and overhead is compared between runs with and without replication while the runs with replication use the double amount of nodes.

| nodes | original runtime [s] | replication runtime [s] | overhead |
|---|---|---|---|
| 4-8 | 2058 | 2069 | 0.53% |
| 8-16 | 1097 | 1311 | 19.51% |

test the load balancing effect of replication. With this method, instead of calling the *act* method of the local actor sequentially, they are executed in parallel through OpenMP tasks. Now, the actors in the simulation can be slowed down while the *DemulActs* and *MulActs* are allowed to execute normally. This is achieved by slowing down the actors in the simulation within their *act* method for five calls with a waiting time of $5000ns$. This time was chosen empirically to achieve an imbalance of a runtime increase factor of around 2 during the execution of the run without replication compared to the previous baseline. If this imbalance increased the runtime by a much larger amount or only insignificantly, the true load balancing effect of the replication would not appear. For a very small overhead, the replicas initially creating an overhead factor of 2 cannot surpass the factor coming from imbalances. In the case of a larger overhead the original actor and the replicas would spend most of their time waiting, which would cause the load balancing to be negligible in comparison.

Using this method, the actors are now able to execute after waiting for a short time instead of being suspended completely during slowdown. This prevents the channels from overflowing too easily which is the reason the runs with OpenMP again are able to use the original channel size of 128. The experiments in Figure 5.6 are executed under similar circumstances as the tests in Figure 5.1 with a grid size of $8000 \times 8000$. The only difference here is the usage of the previously defined imbalanced scenario and the utilization of OpenMP.

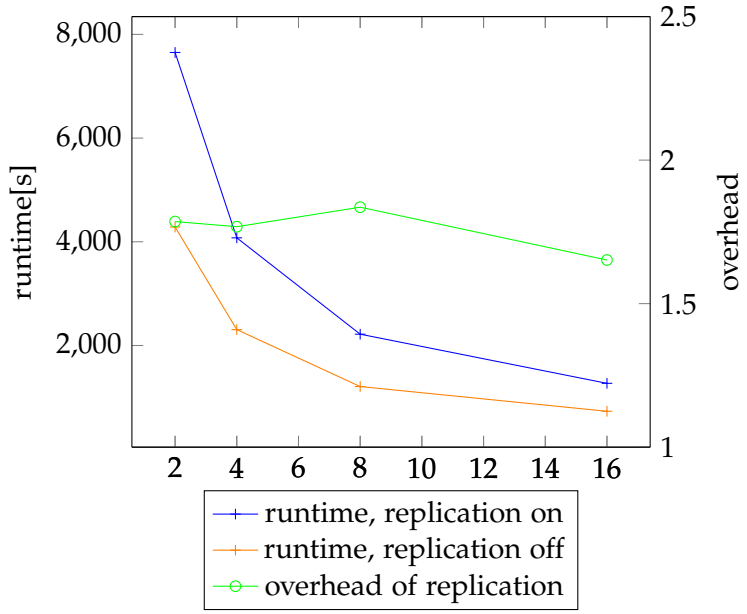



Figure 5.6.: Results of a run with an imbalanced version of Pond with a grid size of $8000 \times 8000$ using 4-16 nodes. Here, OpenMP parallelisation is used within each rank. The end time of the simulation is set to 4 with a patchsize of $250 \times 250$. The orange plot represents the runs without replication while the blue plot shows the runtimes with replication. The green plot represents the overhead induced by replication.

In these experiments, the overhead induced by replication consistently stays below a factor of 2 within the range between 1.652 and 1.836. While the cost of replication is still not fully mitigated in this scenario, there is a notable difference to balanced runs using replication. With the replication inducing an initial minimum overhead factor of 2, it is safe to say that these experiments benefit from a certain degree of load balancing.

Table 5.4.: Runs of imbalanced Pond in an execution using OpenMP multi-threading. The run uses a grid size of $8000 \times 8000$ with a patch size of $250 \times 250$ and an endtime of 4. The runtime and overhead is compared between runs with and without replication while the runs with replication use the double amount of nodes.

| nodes | original runtime [s] | replication runtime [s] | overhead |
|---|---|---|---|
| 2-4 | 4284 | 4076 | -4.85% |
| 4-8 | 2306 | 2218 | -3.82% |
| 8-16 | 1208 | 1270 | 1.05% |

When scaling the nodes in order to make up for the redundancy as in Table 5.4, the overhead in the unbalanced scenario is consistently smaller than in a balanced run. This effect is likely caused by a certain degree of load balancing achieved through replication.

Additionally, when analyzing these run with and without replication through Intel Vtune Amplifier, it is visible that the approach without load balancing is stuck in $upcxx :: barrier()$ calls for more than half of the execution time waiting for other ranks. Meanwhile, in a run with replication, most of the execution time is actually spend within the actors' *act* methods while only a negligible amount execution time is spent waiting.

All in all, while the load balancing effect of replication does not make the redundant run faster than the baseline, the expected effect can be observed by a certain amount.

# 6. Conclusion and Future Work

In this thesis, a replication approach of actors was developed for the actor model. This extension was designed to investigate two possible advantages of using double redundancy compared to the original actor structure. Firstly, the redundant computations were used to check a program run for internal errors. In particular, this approach was proposed to detect SDCs that might otherwise go unnoticed. By checking the messages sent by each original actor and its replicas, it was possible to detect all inserted errors in the test cases while also not reporting any false positives. While these results are particularly accurate, in the case of pond, some errors could have been ignored due to the error-resistant nature of the application.

Secondly, a load balancing approach based on redundancy was proposed. Placing the original actor and its replica on separate ranks, this model hoped to always have one of the two actors produce results even when the other one is slowed down. While this effect was observed in the performed tests on an unbalanced scenario, the load balancing did not have an effect large enough to justify the usage of replication to improve the program runtime. Due to large overhead incurred during the message forwarding in this implementation, the runtimes using replication were not able to surpass the baseline without replication even in a very unbalanced scenario.

Working on actor replication in the future, there is a number of additions that possibly could enhance its functionality or improve the runtimes.

## 6.1. Internal Data Checks

One possible improvement to this model of replication within *Actorlib* would be to perform internal data checks for each actor. In this thesis, the replicated actor is only used in order to perform error checks during an actor's communication. This might lead to problems in cases where an error does not spread outside of an actor. One example for this is an error insertion into the middle of pond while not simulating the scenario long enough for the error to spread to the edges of the block. Therefore, the error is not caught in the *DemulAct*.

This problem might be solved by adding a channel between the original actor and its replica to the *ActorGraph*. Through this channel, the actors regularly send their internal data to their replica or original, which in turn can compare its own state to the received data. Using this method, internal SDCs that do not spread to other actors can be detected. Additionally, this data could possibly be used to restore the state of corrupted actors in case of SDCs.

The major downside of this method is the large amount of data that has to be transferred between the actors in order to achieve the wanted results. Sending the entire data of all actors could be very expensive. This is especially the case since the original actor and its replica, in this implementation, are put on separate ranks. Transferring these large amounts of data between ranks is likely an unfeasible approach for error detection. Instead, maybe only a hash of the actor could be sent to the corresponding redundant actor.

## 6.2. Error Range

While this implementation of error detection using hash values has a high accuracy, some small errors might not influence the results enough to be relevant. Using this thesis' implementation, these errors are not distinguished from large data corruptions. Especially in large-scale computing, some small numerical errors or errors due to parallelism might also cause some indeterminism in the actors, which in turn can be interpreted as a SDC by the error checking algorithm.

In cases where these errors should not be detected, a small range could be defined within which errors are deemed acceptable. This might be done similarly to the method from [3]. One change needed to implement this is the storage of the full received messages instead of their hash values. When a deviance is found between two hash values of messages, the distance between the actual message values cannot be evaluated anymore. One possible problem of this addition might therefore be the increase in memory cost for the program.

## 6.3. Double Replication for Error Correction

In this thesis, the double redundancy provided can only detect errors but then does not introduce a way of handling them. One possible approach for this could be to use triple redundancy and perform error correction similarly to the implemented detection.

Doing this, the *MulAct* has to be expanded to accept messages from an original and two replicas instead of one. Then, when a deviance is detected, the results coming from two out of the three actors is likely the correct message to be forwarded while the third message is discarded. Yet, the corrupted actor in itself cannot continue its computations since the error could have occurred anywhere within the actor and spread to the communication. Therefore, without introducing a method to rollback an actor to a working state, only one error can be detected for each actor, which then has to be stopped. After stopping the actor with an error, the other two actors could still use double redundancy in order to detect further errors without the possibility of a correction.

All in all, it is important to keep the fact in mind that SDCs are very unlikely to occur. Therefore, it might be enough to only correct one error for each actor with the probability of one actor experiencing multiple data corruptions in one run being fairly low.

## 6.4. Combination with Error Handling

Alternatively to correcting the errors with redundancy, it might be possible to combine the introduced double redundancy with other methods of error handling introduced in Chapter 2.

The approach that seems most promising in combination with error detection through replication is checkpointing. Here, in regular intervals, every actor writes checkpoints into persistent storage. Then, whenever a SDC is detected through the replicated actors, the original actor and its replica revert to the last checkpoint. When doing this, the messages received by the actors would have to be stored in order to avoid the actors' input being lost.

Alternatively, all actors could be reset to a checkpoint. Here, it is important to make sure that no actor returns to a checkpoint that is ahead in time to any other actor's last checkpoint. If that was the case, one actor might resend a message that the actor further ahead already received before the reset. This, in turn, can cause errors or even program crashes.

## 6.5. Alternatives to Multiplexing

In order to solve the problem of the large overhead incurred by the *MulActs* or *DemulActs* another form of message passing between actors might be worth a consideration.

For this, a similar method to the one in *RedMPI* might achieve good results. Under the assumption that every actor in the *ActorGraph* is replicated once, there is no need to forward one message to one receiver or multiple messages to one receiver. Instead, for example, all replicated actors can only be connected to other replicas and the originals send their messages to each other. Meanwhile, the actors send a hash of their messages to the other actor, replica or original. This hash value can then be used for error detection. Yet, this approach is not able to perform load balancing like the model in this thesis does.

Instead of sending hash values, sending all messages double to the receiving original actor and its replica might also be a possible way of approaching the channels using replication. In this case, each actor receives every message two times and can do the error detection or load balancing when receiving these messages. Regardless, this method also only works if every actor was replicated which takes away the user's opportunity to implement other approaches like a dynamic replication.

## 6.6. Dynamic Channel Size

One possible problem of *Actorlib* is the limited channel size connecting the ports. When the library's user creates an actor, they have to specify the sizes of the connected channels during compilation, after which it cannot be changed anymore. In order to avoid the problem of possibly overflowing channels, the channel size could be dynamically changed during the program execution depending on how much space is needed.

The fixed channel size particularly turned out to create problems in some very unbalanced scenarios. Here, if one actor keeps writing into a channel while another connected actor is not able to execute due to imbalances a channel overflow crashes the application.

# List of Figures

# List of Tables

# Bibliography


[1] G. A. Agha. *Actors: A model of concurrent computation in distributed systems.* Tech. rep. Massachusetts Inst of Tech Cambridge Artificial Intelligence Lab, 1985.

[2] P. Bannon, G. Venkataramanan, D. D. Sarma, and E. Talpes. "Computer and redundancy solution for the full self-driving computer." In: *2019 IEEE Hot Chips 31 Symposium (HCS)*. IEEE Computer Society. 2019, pp. 1–22.

[3] E. Berrocal, L. Bautista-Gomez, S. Di, Z. Lan, and F. Cappello. "Lightweight silent data corruption detection based on runtime data analysis for HPC applications." In: *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. 2015, pp. 275–278.

[4] A. Breuer and M. Bader. "Teaching parallel programming models on a shallow-water code." In: *2012 11th International Symposium on Parallel and Distributed Computing*. IEEE. 2012, pp. 301–308.

[5] P. G. Bridges, K. B. Ferreira, M. A. Heroux, and M. Hoemmen. "Fault-tolerant linear solvers via selective reliability." In: *arXiv preprint arXiv:1206.1390* (2012).

[6] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. "Toward Exascale Resilience: 2014 update." In: *Supercomputing Frontiers and Innovations* 1.1 (2014). DOI: 10.14529/jsfi140101.

[7] D. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel. "An Analysis of Resilience Techniques for Exascale Computing Platforms." In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 52017, pp. 914–923. ISBN: 978-1-5386-3408-0. DOI: 10.1109/IPDPSW.2017.41.

[8] S. Di and F. Cappello. "Adaptive impact-driven detection of silent data corruption for HPC applications." In: *IEEE Transactions on Parallel and Distributed Systems* 27.10 (2016), pp. 2809–2823.

[9] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar. *Silent Data Corruptions at Scale*.

[10] D. J. Fiala, F. Mueller, C. Engelmann, K. B. Ferreira, R. Brightwell, and R. Riesen. *Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing*. 2012. DOI: 10.2172/1081941.

[11] C. Hewitt, P. Bishop, and R. Steiger. *A universal modular ACTOR formalism for artificial intelligence*.

[12] S. Hukerikar, P. C. Diniz, and R. F. Lucas. "A case for adaptive redundancy for HPC resilience." In: *European Conference on Parallel Processing*. Springer. 2013, pp. 690–697.

[13] D. B. Johnson. *Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing*. New York, NY: ACM, 1988. ISBN: 0897912772.

[14] Leibniz-Rechenzentrum. *CoolMUC-2 documentation*. 2.08.2021.

[15] R. J. LeVeque, D. L. George, and M. J. Berger. "Tsunami modelling with adaptively refined finite volume methods." In: *Acta Numerica* 20 (2011), 211–289. DOI: 10.1017/S0962492911000043.

[16] T. G. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Pearson Education, 2004.

[17] E. Meneses, X. Ni, G. Zheng, C. L. Mendes, and L. V. Kale. "Using Migratable Objects to Enhance Fault Tolerance Schemes in Supercomputers." In: *IEEE Transactions on Parallel and Distributed Systems* 26.7 (2015), pp. 2061–2074. ISSN: 1045-9219. DOI: 10.1109/TPDS.2014.2342228.

[18] A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski. "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System." In: *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 112010, pp. 1–11. ISBN: 978-1-4244-7557-5. DOI: 10.1109/SC.2010.18.

[19] A. Pöppl, S. Baden, and M. Bader. *A UPC++ Actor Library and Its Evaluation On a Shallow Water Proxy Application*. Piscataway, NJ: IEEE, 2018. ISBN: 9781538655412.

[20] A. Poppl, M. Bader, T. Schwarzer, and M. Glass. "SWE-X10: Simulating Shallow Water Waves with Lazy Activation of Patches Using Actorx10." In: *2016 Second International Workshop on Extreme Scale Programming Models and Middlewar (ESPM2)*. IEEE, 112016, pp. 32–39. ISBN: 978-1-5090-3858-9. DOI: 10.1109/ESPM2.2016.010.

[21] S. Roloff, A. Pöppl, T. Schwarzer, S. Wildermann, M. Bader, M. Glaß, F. Hannig, and J. Teich. "ActorX10: an actor library for X10." In: *Proceedings of the 6th ACM SIGPLAN Workshop on X10*. Ed. by C. Fohry and O. Tardieu. New York, NY, USA: ACM, 6022016, pp. 24–29. ISBN: 9781450343862. DOI: 10.1145/2931028.2931033.

[22] *UPC++ v1.0 Programmer's Guide, Revision 2020.10.0*. 9.11.2020.

[23] J. F. Wakerly. "Microcomputer reliability improvement using triple-modular redundancy." In: *Proceedings of the IEEE* 64.6 (1976), pp. 889–895. ISSN: 0018-9219. DOI: 10.1109/PROC.1976.10239.

# A. Dependencies

The following libraries were used in order to build the actor-based replication. The testing was done on the CoolMUC-2 linux cluster of the Leibniz-Rechenzentrum (LRZ).

- **Actorlib** and **Pond**: https://bitbucket.org/apoeppl/actor-upcxx/src/master/
- **UPC++** 2021.3.0: https://bitbucket.org/berkeleylab/upcxx/wiki/Home
- **Intel C++ Compiler** 19.0.5
- **Intel MPI** 2019.7
- **NetCDF** 4.7-hdf5
- **Metis** 5.1.0
- **CMake** 3.16.5
- **Boost** 1.75.0

# B. Scripts

In order to build the library and create submittable scripts for the cluster, a number of scripts have been used from *Actorlib* and changed to support newly added variables. When trying to create the submittable jobscripts for pond, first the pond input parameters have to be set in *set_vars_for_jobgeneration*. Then, the *generator* creates the necessary files and folders. Lastly, the *builder* script builds pond. The resulting jobscripts can then be submitted to the cluster via the *sbatch* command.

```
#!/bin/bash

#job types generate subfolders for every type
_jobtypes=(plain)
_jobtypes=$( IFS=' '; printf '%s' "${_jobtypes[*]}" )
export s_jobtypes=$_jobtypes
#sizes of the total grid SxS
_sizes=(8000)
_sizes=$( IFS=' '; printf '%s' "${_sizes[*]}" )
export s_sizes=$_sizes
#an actor will have patchsize SxS
export patchsize=250
#number of cpus per node
export corecountpernode=28
#how many nodes the job will run
_nodecounts=(8)
_nodecounts=$( IFS=' '; printf '%s' "${_nodecounts[*]}" )
export s_nodecounts=$_nodecounts
#end times for the simulation
_endtimes=(4)
_endtimes=$( IFS=' '; printf '%s' "${_endtimes[*]}" )
export s_endtimes=$_endtimes
#the name of folder to save the work
export workdir=test_group
#the upcxx_install path for cmake, if you have installed somewhere else
↪  then change this
```

```
export UPCXX_INSTALL=~/upcxx-intel-mpp3
#add upcxx path to the path, if you have installed somewhere else then
↪  change it
export PATH=$PATH:~/upcxx-intel-mpp3/bin
_seeds=(15 36 79 178 5972 6529 9231 10492 15637 201392)
_seeds=$( IFS=' '; printf '%s' "${_seeds[*]}" )
export seeds=$_seeds

#!/bin/bash

#predefined sizes

#WARNING
# this script uses the environment variables set in set_vars_for_jobs.sh

basetime=600
#convert_time does not work for every size, check the generator and the
↪  function for the details

#convert from string to array
IFS=' ' read -r -a jobtypes <<< "$s_jobtypes"
#echo ${jobtypes[*]}
IFS=' ' read -r -a sizes <<< "$s_sizes"
#echo ${sizes[*]}
IFS=' ' read -r -a nodecounts <<< "$s_nodecounts"
#echo ${nodecounts[*]}
IFS=' ' read -r -a endtimes <<< "$s_endtimes"
#echo ${endtimes[*]}
IFS=' ' read -r -a seeds <<< "$seeds"


rm -r -f jobscripts
#create jobscripts
if [ -d jobscripts/ ]
then
    echo "jobscripts exists"
else
    echo "create jobscripts"
    mkdir jobscripts
```

```
    chmod -R 775 jobscripts
fi

#check for scratch
if [ -z "$SCRATCH" ]
then
    echo "scratch not set"
    SCRATCH=${PWD}
fi

#create ${workdir} in scratch
if [ -d ${SCRATCH}/${workdir} ]
then
    echo "${workdir} in scracth exists"
else
    mkdir ${SCRATCH}/${workdir}
    chmod -R 775 ${SCRATCH}/${workdir}
fi

cd jobscripts

#convert_time does not work for every size, check the generator and the
↪  function for the details
function convert_time {
    if [ "$size" =  "2000" ]
    then
        limit=$(( limit / 16 ))
    fi
    if [ "$size" =  "4000" ]
    then
        limit=$(( limit / 4 ))
    fi

    if [ "$nodecount" = "1" ]
    then
        limit=$(( limit * 2 ))
    fi
    if [ "$nodecount" = "4" ]
    then
```

```
    limit=$(( limit / 2 ))
fi
if [ "$nodecount" = "8" ]
then
    limit=$(( limit / 4 ))
fi
if [ "$nodecount" = "16" ]
then
    limit=$(( limit / 8 ))
fi
if [ "$nodecount" = "32" ]
then
    limit=$(( limit / 16 ))
fi

if [ "$patchsize" = "250" ]
then
    limit=$(( limit * 4))
fi

limit=$(( limit * endtime + 1800))

hours=$(( limit / 3600 ))
mins=$(( limit % 3600 ))
secs=$(( mins % 60 ))
mins=$(( mins / 60 ))

if [[ "$hours" -ge "0" && "$hours" -le "9" ]]
then
    hours=0${hours}
fi
if [[ "$mins" -ge "0" && "$mins" -le "9" ]]
then
    mins=0${mins}
fi
if [[ "$secs" -ge "0" && "$secs" -le "9" ]]
then
    secs=0${secs}
fi
```

```
}

#generate folders for each task
for size in ${sizes[@]}
do
    for nodecount in ${nodecounts[@]}
    do
        for endtime in ${endtimes[@]}
        do
            for job in ${jobtypes[@]}
            do
               for seed in ${seeds[@]}
               do
                str=pond-${job}-${nodecount}-${size}x${size}
                -${endtime}-${seed}
                if [ -d ${str} ]
                then
                    #rm -r ${str}
                    echo "old results for "${str}" exists, did not touch"
                else
                    mkdir ${str}
                fi

                cd ${str}

                scriptname=pond-${job}.sh

                #write the modified base script
                if [ -f "$scriptname" ]
                then
                    rm $scriptname
                fi

cat <<EOF >$scriptname
#!/bin/bash
#SBATCH -J ${str}
#SBATCH -o ${SCRATCH}/${workdir}/${str}/%x.%j.out
#SBATCH -e ${SCRATCH}/${workdir}/${str}/%x.%j.err
#SBATCH -D ./
```

```
#Notification and type
#SBATCH --mail-type=end,fail,timeout
#SBATCH --mail-user=<e-mail address>
# Wall clock limit:
#SBATCH --time=${hours}:${mins}:${secs}
#SBATCH --no-requeue
#Setup of execution environment
#SBATCH --export=NONE
#SBATCH --get-user-env
#SBATCH --clusters=cm2
#SBATCH --partition=cm2_std
#SBATCH --nodes=${nodecount}
#SBATCH --ntasks-per-node=${corecountpernode}

module unload intel-mpi
module load slurm_setup
module load intel
module load intel-mpi/2019-intel
module load netcdf-hdf5-all/4.7_hdf5-1.10-intel19-serial
module load metis/5.1.0-intel19-i64-r64

export UPCXX_INSTALL=~/upcxx-intel
export PATH=\$PATH:~/upcxx-intel/bin
export GASNET_PHYSMEM_MAX='32 GB'
#41 GB is the max number in mpp2 (both _inter and _tiny)

upcxx-run -n ${corecountforjob} -N ${nodecount} -shared-heap 512MB
↪  ./pond-${job} -x ${size} -y ${size} -p ${patchsize} -c 10 --scenario
↪  2 -o ${SCRATCH}/${workdir}/${str}/out/out -e ${endtime} -a ${seed}

cat > ${SCRATCH}/${workdir}/${str}/finished.txt

EOF
                chmod 775 $scriptname
                echo "Generated "$scriptname
                cd ..
            done
        done
    done
```

```
done
done

cd ..
echo ${pwd}
echo "done generating script folders"

echo "generate folders in scratch"
for size in ${sizes[@]}
do
    for nodecount in ${nodecounts[@]}
    do
        for endtime in ${endtimes[@]}
        do
            for job in ${jobtypes[@]}
            do
                for seed in ${seeds[@]}
                do
                str=pond-${job}-${nodecount}-${size}x${size}
                -${endtime}-${seed}
                if [ -d ${SCRATCH}/${workdir}/${str} ]
                then
                    echo "old results for "${SCRATCH}/${workdir}/${str}"
                     ↪  exists, did not touch"
                else
                    mkdir ${SCRATCH}/${workdir}/${str}
                    mkdir ${SCRATCH}/${workdir}/${str}/out
                fi
                chmod -R 775 ${SCRATCH}/${workdir}/${str}
            done
        done
    done
done
done

#!/bin/bash

#script for building and copying right executables
```

```
module load boost
module load intel
module load intel-mpi
module load netcdf-hdf5-all
module load metis/5.1.0-intel19-i64-r64
module load cmake/3.16.5

make clean
rm CMakeCache.txt
rm -r CMakeFiles
rm cmake_install.cmake
rm Makefile

export UPCXX_INSTALL=~/upcxx-intel
#JOBTYPES is read as an environment variable, the user has to make sure
 ↪  that the needed jobtypes are compiled!!!!

#no loop for jobtypes here because every job type will have a different
 ↪  set of arguments so, copy by yourself
cmake . -DCMAKE_C_COMPILER=mpiicc -DCMAKE_CXX_COMPILER=mpiicpc
 ↪  -DCMAKE_PREFIX_PATH=${UPCXX_INSTALL} -DENABLE_FILE_OUTPUT=OFF
 ↪  -DBUILD_RELEASE=ON \
-DENABLE_LOGGING=OFF -DENABLE_O3_UPCXX_BACKEND=ON
 ↪  -DENABLE_PARALLEL_UPCXX_BACKEND=ON -DENABLE_MEMORY_SANITATION=OFF
 ↪  -DIS_CROSS_COMPILING=OFF \
-DPRINT=OFF -DINVASION=OFF -DRANKMIG=OFF -DTIME=OFF -DANALYZE=OFF
 ↪  -DTRACE=OFF -DMIGRATION=0 -DREPLICATION=ON -DBOTTLENECK=OFF
 ↪  -DINBALANCE=ON -DERRORMIDDLE=OFF -DERROREDGE=OFF

make actorlib -j 16
make pond -j 16
#pond-* should be same as pond-${jobtypes[@]} that is an environment
 ↪  variable

if [ -d jobscripts ]
then
    echo "jobscripts already exists"
else
    echo "no sense if jobscripts are not generated"
```

```
    exit
fi

cd jobscripts

#copy executables to alreay created files by the generator.sh
for t in ${jobtypes[@]}
do
    for i in pond-${t}-*/
    do
        echo $i
        cp ../pond-${t} "$i"
    done
done

cd ..
```

# C. Random Seeds

Here, the random seeds used for the 100 error detection tests are listed. For every random seed, it is noted whether or not an error was inserted into the middle of a block, if the replication-based error detection reported an error and whether or not the program crashed.

| Random seed | Error inserted? | SDC detected? | Program crashed? |
|---|---|---|---|
| 10492 | Yes | Yes | No |
| 15637 | Yes | Yes | No |
| 201392 | Yes | Yes | No |
| 5972 | Yes | Yes | No |
| 79 | Yes | Yes | No |
| 15 | Yes | Yes | No |
| 178 | No | No | No |
| 36 | Yes | Yes | No |
| 6529 | Yes | Yes | No |
| 9231 | Yes | Yes | No |
| 142 | No | No | No |
| 5643 | No | No | No |
| 82930480 | Yes | Yes | No |
| 1563735 | No | No | No |
| 623 | Yes | Yes | No |
| 836479 | Yes | Yes | No |
| 17382 | No | No | No |
| 6321230 | Yes | Yes | No |
| 83721 | No | No | No |
| 18291 | No | No | No |
| 632712 | Yes | Yes | No |
| 8372173 | No | No | No |
| 183722 | Yes | Yes | No |
| 635212 | No | No | No |
| 849327382 | No | No | No |

| Random seed | Error inserted? | SDC detected? | Program crashed? |
|---|---|---|---|
| 2138126 | No | No | No |
| 638291 | Yes | Yes | No |
| 86473284 | Yes | No | No |
| 23712684 | Yes | Yes | No |
| 6426382 | Yes | Yes | No |
| 9123846 | No | No | No |
| 2739123 | Yes | No | No |
| 653821 | No | No | No |
| 9263718 | No | No | No |
| 2819827 | Yes | Yes | No |
| 676342 | No | No | No |
| 9283 | No | No | No |
| 29837 | Yes | Yes | No |
| 7231 | No | No | No |
| 93747 | Yes | Yes | No |
| 3273621 | Yes | Yes | No |
| 73 | No | No | No |
| 94371927 | Yes | Yes | No |
| 4325562 | No | No | No |
| 7312273 | No | No | No |
| 9537 | Yes | Yes | No |
| 45912 | No | No | No |
| 736299237 | No | No | No |
| 98 | No | No | No |
| 4636723 | Yes | Yes | No |
| 78216 | No | No | No |
| 983628 | No | No | No |
| 5487 | No | No | No |
| 7895862 | No | No | No |
| 9837261 | No | No | No |
| 1002 | Yes | Yes | No |
| 634726 | Yes | Yes | No |
| 12371 | No | No | No |
| 637772 | No | No | No |
| 1237919 | Yes | Yes | No |
| 638282 | No | No | No |
| 127399 | Yes | Yes | No |
| 726381 | Yes | Yes | No |

| Random seed | Error inserted? | SDC detected? | Program crashed? |
|---|---|---|---|
| 1372920 | No | No | No |
| 7372828 | Yes | No | No |
| 173128 | No | No | No |
| 738247 | Yes | Yes | No |
| 18737 | Yes | Yes | No |
| 8211 | Yes | Yes | No |
| 1932857 | Yes | Yes | No |
| 828391 | Yes | Yes | No |
| 21327 | Yes | Yes | No |
| 82939 | No | No | No |
| 23042 | No | No | No |
| 83201 | No | No | No |
| 23671 | Yes | No | No |
| 83718 | No | No | No |
| 2617 | Yes | Yes | No |
| 838291 | Yes | Yes | No |
| 281273681 | No | No | No |
| 83928 | No | No | No |
| 31278417 | No | No | No |
| 908290 | Yes | No | No |
| 3204972398 | Yes | Yes | No |
| 91 | No | No | No |
| 3622728 | No | No | No |
| 912387 | No | No | No |
| 3712683 | No | No | No |
| 91299 | No | No | No |
| 382929 | No | No | No |
| 92283 | Yes | Yes | No |
| 3829418 | No | No | No |
| 92321 | Yes | Yes | No |
| 38421 | Yes | Yes | No |
| 938223 | No | No | No |
| 3929891 | No | No | No |
| 9483 | No | No | No |
| 4773 | No | No | No |
| 98879 | No | No | No |
| 632212 | No | No | No |