# Extending the MPI Backend of X10 by Elasticity

Marco Bungart
University of Kassel
Germany
marco.bungart@uni-kassel.de

Claudia Fohry
University of Kassel
Germany
fohry@uni-kassel.de

## ABSTRACT

The X10 parallel programming language supports resiliency and elasticity. Resiliency denotes the capability to tolerate permanent node failures, and elasticity denotes the capability to add new nodes to a running computation. X10 is implemented with different backends. The MPI backend supports resiliency, but not elasticity. This poster describes an extension that supplements this feature. The extension deploys MPI's client/server routines and a network socket.

## CCS CONCEPTS

• **Computing methodologies** → *Parallel programming languages*; *Distributed programming languages*;

## KEYWORDS

MPI, X10, Elasticity, Client/server routines

**ACM Reference format:**
Marco Bungart and Claudia Fohry. 2017. Extending the MPI Backend of X10 by Elasticity. In *Proceedings of European MPI User's Group Meeting, Chicago, Illinois, USA, September 2017 (EuroMPI),* 2 pages.
https://doi.org/

## OVERVIEW

X10 is a PGAS language that has been developed by IBM [2]. Computational resources, together with a memory partition form a *place*. Each place can access each memory partition, but access to the local partition is faster. Places are numbered consecutively, starting with 0. X10 has a Java-like syntax and provides constructs for intra-place parallelization and inter-place communication. Intra-place parallelization deploys tasks, which are called activities. Inter-place communication is based on active messages, i.e., an activity that runs on one place may spawn an activity on another. Thereby it may pass data copies. The only means to access remote data are active messages and RMA functions. An X10 program always starts with a single activity on place 0. Later it incorporates the other places by invoking activities there.

X10 supports resiliency and elasticity since version 2.4.1 [2, 4] . Resiliency denotes the capability to tolerate permanent place failures, and elasticity denotes the capability to incorporate new places into a running computation. The new places are provided "empty", i.e., without a user program running on them. Therefore, the existing places must detect the new ones and invoke activities there. Detection is based on inquiry functions. For instance, X10 provides a function `places()`, which returns a list of places. After additions, it returns an updated list, so that the difference can be recognized.

X10 programs can be compiled to Java (called Managed X10) or to C++ (called Native X10). Managed X10 implements resiliency and elasticity in a sockets-based backend [2]. Native X10 implements resiliency in an MPI-based backend [5], based on ULFM [3]. Other backends do not yet support resiliency or elasticity.

This paper describes an extension of the resilient MPI backend by elasticity. From a user's point of view, elasticity is switched on by setting environment variable `X10_ELASTIC=1` before compilation. When the program is started (via `mpirun`), it opens a port and prints information to the console. The user should then set environment variable `X10_JOIN_EXISTING=<hostname>[:<port>]`. Thereupon, all subsequently started X10 programs dispense their places to the original program instead of invoking an executable. The executable must nevertheless be provided, and it must coincide with the original one. On the new places, the X10 runtime is started and the executable's classes are loaded.

Our implementation of elasticity has different aspects: First, the original program opens a port and listens for incoming join requests. Second, an extended world communicator is built with the help of MPI's client/server routines, and `MPI_COMM_WORLD` is replaced by this communicator. A major difficulty herein is the need to finish all communication on the original communicator before establishing the extended one. Finally, we must ensure a correct numbering of the X10 places.

In the remaining paragraphs of this paper, we provide background on the MPI backend of X10, discuss the different aspects of our implementation, and mention related work.

*X10's resilient MPI backend.* Like X10, the resilient MPI backend is open-source [1]. It is based on Open MPI ULFM (version 1.7.1) [3], which does not support one-sided communication and requires threading level `MPI_THREAD_SERIALIZED`.

In the backend, MPI ranks coincide with X10 place numbers. The backend provides interface functions to be called from X10. They include:

- `x10rt_net_send(p, msg)`: sends message `msg` to place $p$,
- `x10rt_net_send_get(p, data)`: requests `data` from place `p` (required for RMA implementation), and
- `x10rt_net_probe()`: is called regularly and receives all pending messages at the respective place.

The interface functions are implemented with MPI using a distributed data structure called `global_state`. At each place, it contains:


*EuroMPI, September 2017, Chicago, Illinois, USA*

ACM ISBN …$15.00
https://doi.org/

- a threadsafe list of reusable MPI-requests, called `free_req`
- threadsafe lists of outstanding requests, called `pending_sends` and `pending_recvs`, and
- a copy of `MPI_COMM_WORLD`, called `mpi_comm`.

Internally, `x10rt_net_send()` calls `MPI_Isend()`, using a request from `free_req`. Function `x10rt_net_probe()` internally calls `MPI_Iprobe()`, followed by `MPI_Irecv()` and `MPI_Test()` if messages are pending. After the respective calls, outstanding requests are inserted into `pending_sends` and `pending_recvs`.

The implementation of `x10rt_net_send_get(p, data)` uses auxiliary functions `get_req()` and `get_data()`. They request data from another place, and deliver these data, respectively. These two steps reuse their MPI requests at source and destination, respectively.

Collective operations are not called on `mpi_comm`, but X10's team operations are implemented with other communicators.

*Providing a Connection Point.* In the following, we denote the original X10 program as *server*, and the joining program as *client*. An obvious approach to connection establishment would be MPI's client/server routines. Unfortunately, they involve the function `MPI_Comm_accept()`, which is blocking. Therefore its use would prevent the server processes from advancing the application. Performing the call in a separate thread does not help either, because of the `MPI_THREAD_SERIALIZED` setting. Therefore, we decided to use a network socket for establishing the first contact between client and server. A network socket has the additional advantage of supporting a timeout, which simplifies program termination. It is opened in a separate thread at process 0 of the server.

*Connection establishment in MPI.* While the network port is used for connection establishment, the integration of the new resources into MPI requires an MPI port. This port is opened by the same thread of process 0 as the network port. Before that, however, all communication on `MPI_COMM_WORLD` must be terminated as explained in the next paragraph. `MPI_Comm_accept` is called collectively in the server. Thereafter, process 0 informs the client, which in turn collectively calls `MPI_Comm_connect()`. If any of the calls returns with an error, the corresponding communicator is repaired and the accept / connect calls are repeated. Failure management relies on ULFM functions such as `MPI_Comm_agree` and `MPI_Comm_shrink`.

*Terminating communication on the old communicator.* In the MPI backend of X10, the function `x10rt_net_probe()` refers to the `global_state` and listens on `mpi_comm` only. If we would update `mpi_comm` while some messages on the old communicator are still in flight, these messages would never be received. Thus, we need to ensure that no messages on the old `mpi_comm` are pending.

We denote a state without pending messages as *silent*. To reach silent state, process 0 of the server sends a message `stop_send` to all server processes and waits for their acknowledgements. On reception of this message, a process blocks `free_req`, to prevent new network operations from starting. Still, ongoing operations can send and receive data, when they reuse requests. Therefore, our extension additionally replaces `MPI_Isend` by `MPI_Issend()` so that `MPI_Test()` does not return before the message has been recognized at the receiver.

This way, we can be sure to have reached silent state, if on all server ranks both `pending_sends` and `pending_recvs` are empty. Each processes sends a `rdy_join` message to process 0 when its queues are empty. Thereafter, it waits for an incoming `do_accept`, while continuing to call `x10rt_net_probe()` to finish two-step communication on the old communicator. This is necessary, since some other processes might be waiting for a response.

When rank 0 has received a `rdy_join` message from each rank, it notifies the server processes by sending message `do_accept`. The processes thereupon call `MPI_Comm_accept()`, as explained before.

*Rebuilding the Communicator.* Upon return of `MPI_Comm_accept()` and `MPI_Comm_connect()`, all client and server processes hold an intercommunicator to the other group. We convert it into an intracommunicator with function `MPI_Intercomm_merge()`. It places server processes before client processes so that, outside failures, server processes keep their original rank.

Unfortunately, the new communicator does not include failed processes. X10, in contrast, stays with the original place numbering, leaving gaps for failed places. We resolve this inconsistency with the help of an additional field `mpi_group` in `global_state`. It is initialized with the group of the original `MPI_COMM_WORLD`. After the call to `MPI_Intercomm_merge()`, we unify `mpi_group` with the group of the new communicator. X10 place numbers are computed with the help of MPI function `MPI_Group_translate_ranks`.

Thereafter, `free_req` is un-blocked. The places can resume communication, now referring to the extended set of places. In particular, the old places can invoke X10's inquiry functions to detect the new places, and start activities there.

*Related work.* Implementing elasticity with MPI's client/server routines has already been studied in [6]. That work considered a master/slave application, for which it was sufficient to construct a set of pairwise communicators. The X10 backend, in contrast, requires an updated `MPI_COMM_WORLD`. There is a common restriction in both papers: If the `MPI_Comm_accept()` call is open and the client fails before invoking `MPI_Comm_connect()`, the server will block forever. This problem appears to be intrinsic to MPI.

*Acknowledgements.* Special thanks go to Sara Salem Hamouda for providing an introduction to the design of X10's MPI backend. This work is partially supported by the Deutsche Forschungsgemeinschaft, under grant FO 1035/5-1.

## REFERENCES

[1] [n. d.]. X10 Github Repository. ([n. d.]). https://github.com/x10-lang
[2] [n. d.]. X10 Home Page. ([n. d.]). http://x10-lang.org/
[3] Wesley Bland. 2012. User Level Failure Mitigation in MPI. In *Proc. Euro-Par.* Springer LNCS 7640, pp. 499–504.
[4] David Cunningham, David Grove, Benjamin Herta, et al. 2014. Resilient X10: Efficient failure-aware programming. In *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. pp. 67–80.
[5] Sara S. Hamouda, Benjamin Herta, Josh Milthorpe, David Grove, and Olivier Tardieu. 2016. Resilient X10 over MPI User Level Failure Mitigation. In *Proc. ACM SIGPLAN X10 Workshop.* pp. 19–23.
[6] Claudia Leopold, Michael Süß, and Jens Breitbart. 2006. Programming for Malleability with Hybrid MPI-2 and OpenMP - Experiences with a Simulation Program for Global Water Prognosis. In *Proc. European Conference on Modelling and Simulation.* pp. 665–670.