DEGAS: Dynamic Global Address Space programming environments

Final Report from The University of Texas at Austin
Mattan Erez, Co-Principal Investigator[1]
The University of Texas, Austin, Texas

Katherine Yelick, Principal Investigator[2]
Lawrence Berkeley National Laboratory, Berkeley, California

Vivek Sarkar, Co-Principal Investigator[3]
Rice University, Houston, Texas

James Demmel, Co-Principal Investigator[4]
University of California, Berkeley, California

Abstract: The Dynamic, Exascale Global Address Space programming environment (DEGAS) project will develop the next generation of programming models and runtime systems to meet the challenges of Exascale computing. Our approach is to provide an efficient and scalable programming model that can be adapted to application needs through the use of dynamic runtime features and domain-specific languages for computational kernels. We address the following technical challenges:

Programmability: Rich set of programming constructs based on a Hierarchical Partitioned Global Address Space (HPGAS) model, demonstrated in UPC++.
Scalability: Hierarchical locality control, lightweight communication (extended GASNet), and efficient synchronization mechanisms (Phasers).
Performance Portability: Just-in-time specialization (SEJITS) for generating hardware-specific code and scheduling libraries for domain-specific adaptive runtimes (Habanero).
Energy Efficiency: Communication-optimal code generation to optimize energy efficiency by reducing data movement.
Resilience: Containment Domains for flexible, domain-specific resilience, using state capture mechanisms and lightweight, asynchronous recovery mechanisms.
Interoperability: Runtime and language interoperability with MPI and OpenMP to encourage broad adoption.

[1]UT-PI Email: mattan.erez@utexas.edu; Phone: (512) 471-7846
[2]PI Email: kayelick@lbl.gov; Phone: (510) 495-2431
[3]Rice-PI Email: vsarkar@rice.edu; Phone: (713) 348-5304
[4]UCB-PI Email: demmel@cs.berkeley.edu; Phone: (510) 643-5386

Main Accomplishments

The University of Texas at Austin component of the DEGAS project focused on resilience research. In particular, we had set XXX main objectives. We list these objectives below and then briefly highlight our main findings. We have accomplished all but the final objective, on which we are still working (without the use of any grant funds). We have:

1. We developed a general resilience model for partitioned global address space (PGAS) programming models. This model includes a new resilience protocol that ensures consistent recovery for uncoordinated per-node and per-task resilience actions, along with possible tradeoffs and simplifications that balance overheads with expressiveness. The model has been published as a technical report [?] and we are working toward a peer-reviewed publication. To the best of our knowledge this is the first general PGAS resilience model.
2. We specified the programming abstractions and interfaces for expressing resilience concerns in hierarchical-PGAS languages, which make use of our resilience model. These were published in a technical report [?] and as part of the documentation of the Containment Domains prototype runtime system [2].
3. We developed and released a prototype resilience component for UPC++, based on the interfaces and protocols mentioned above and the Containment Domains abstractions for PGAS [?].
4. We are working on evaluating the resilient UPC++ prototype and publishing the results. The delay in achieving this objective relate to our decision to implement the Containment Domains resilience runtime both on top of UPC++ and for UPC++ applications and delays in simply compiling and correctly running existing UPC++ applications.

Findings Highlights

Our research and design is based on the idea of Containment Domains (CDs) [3], which extend prior resilience models in two important ways for the DEGAS project. First, CDs are designed to be hierarchical and explicit to the programmer. Second, CDs include support for both *strict CDs*, which allow uncoordinated preservation and uncoordinated recovery only between groups of non-communicating tasks (though with a hierarchy, and *relaxed CDs* that can be completely uncoordinated. Within the CD framework, the tradeoff between additional bookkeeping required for relaxed CDs vs. the additional recovery time for strict CDs is flexible and tunable.

Our model and protocol for PGAS CDs is, to the best of our knowledge, the first to offer uncoorinated resilience in the general PGAS setting for any properly-synchronized application without data races (more precisely, without races between local and remote accesses). The closes prior work [1] was developed for one-sided MPI and is not general enough for the PGAS models proposed in the DEGAS project, such as UPC++. The main effort was focused on designing and implementing relaxed CDs for UPC++ and the protocols needed for the very general UPC++. Our main finding is that the excessive communication logging needed for uncoordinated recovery between communicating PGAS tasks can be curbed if following the two observations below.

Observation I

One observation is that the logging overhead for remote reads/writes is smaller than local reads/writes. Because logging operations are the same for local and remote reads/writes, but remote reads/writes themselves are more expensive than local ones. We only log remote reads and writes, while using remote read/write logs to reconstruct memory to up-to-date states to reexecute local reads/writes.

Two tasks are involved in each remote read/write: one task is the initiator of the remote read or write, and the other task is the one which owns the data of the read/write accesses. For example, shown in Figure 1, task *T0* is the initiator of remote write and task *T1* owns the data of the write access, while task *T2* is the initiator of remote read and task *T3* owns the data of the read access.

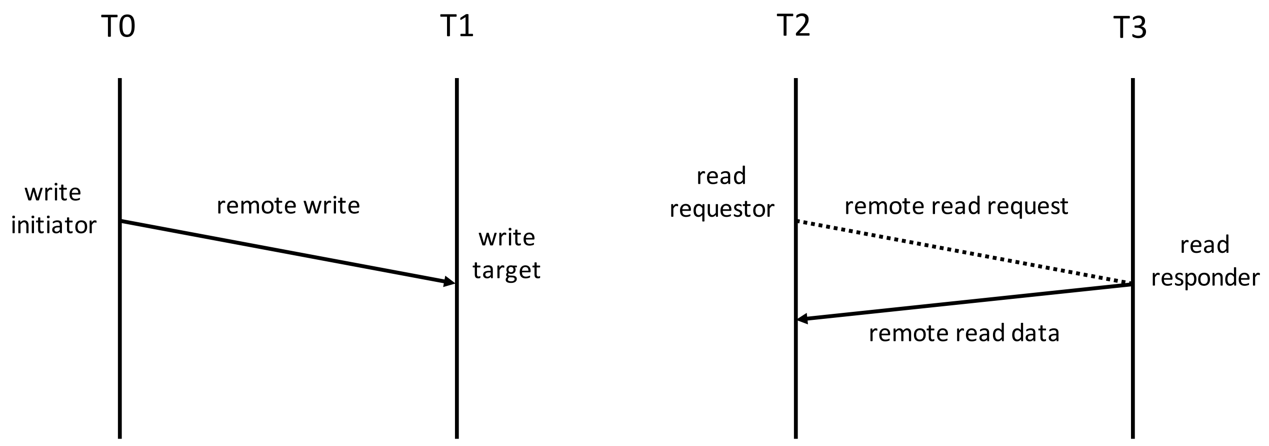

Figure 1: Terminology for remote write (left) and remote read (right)

To make it easier for future discussion, T0 is defined as *write initiator*, T1 as *write target*, T2 as *read requestor*, and T3 as *read responder*.

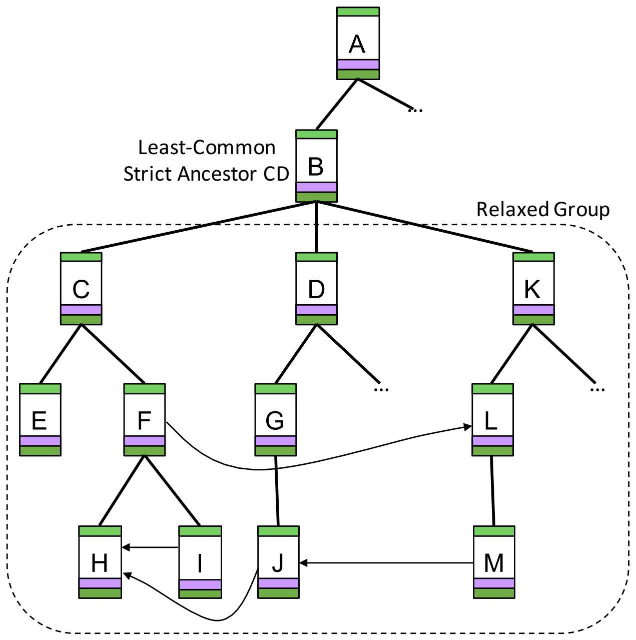

Figure 2: Relaxed Group and Least Common Strict Ancestor (LCSA) CD

One tradeoff for remote read/write logging is which task should log the data: the initiator of read/write or the task who owns the data. We can mimic the sender-side logging mechanisms of MPI two-sided communication, and delibrately put logs in a different task from the application data being logged. Thus, we log writes at the write initiator while logging reads at the read responder. The benefits of this logging scheme is to keep application data and logging data at two different places to survive from catastrophic failures. However, this scheme needs to change every remote read to be an active message so log entries can be generated at the read responder, which may incur high overhead for remote reads.

Instead, we log write at the write-initiator side and to log reads at the read-requestor side. This scheme does not require remote reads to be active messages, but the possibility of one failure causing loss of both application data and read logs must be addressed.

Observation II

Another observation is that data reads/writes themselves may not be real communication, but only real communication should be logged. To help filter out non-communication accesses from logging, we define a *relaxed group* to be the group of relaxed CDs that may communicate with each other. We further define the *least common strict ancestor (LCSA) CD* for a relaxed group to be the lowest-level strict CD that encompass the whole relaxed group. Figure 2 gives an example of a *relaxed group* and its *least common strict ancestor CD*.

Then reads/writes are categorized into three categories, *private*, *temporarily privatized*, and *potentially shared*. The definitions of them are as follows:

Private

A read or write operation is *private* if it accesses data that will not be read or written by any other CD. Note that private addresses may not be used by any other CD until the entire relaxed group, to which the CD belongs, completes, or, may be reused provided any private address is written over before being read again.

Temporarily privatized

A read or write operation is *temporarily privatized* if it accesses data that is: (1) private during the CD execution, (2) was not written by an earlier sibling CD within the same relaxed group, and (3) may be used by other CDs in the same relaxed group after the CD completes.

Potentially shared

A read or write operation is potentially shared if it cannot be classified as either private or temporarily privatized.

All three categories are logical to applications and it does not matter whether reads/writes are local or remote. *Private* reads/writes are not real communication, because only one CD touches the data during its execution. We skip logging for private accesses and reexecute them directly in reexecution. *Potentially shared* data, however, implies potential communication. We log all potentially shared accesses in forward execution, and replay them in reexecution.

The rationale behind *temporarily privatized* accesses is that a common access pattern in HPC applications is that each task works on its local copy of shared data and only commits once at the end. This accessed data is shared between tasks, but private to the task until its final commit. We log temporarily privatized accesses once, at CD complete time, and all temporarily privatized accesses are reexecuted in a shadow memory during recovery. Temporarily privatized accesses need to reexecute in shadow memory, because other non-failed CDs that are executing may touch the same memory regions used by a failed CD that is reexecuting, yet the reexecuting CD may update and reuse those addresses as it reexecutes.

Logging Mechanisms Details

To summarize, we log potentially shared remote reads at the read-requestor side and to log potentially shared remote writes at the write-initiator side. In reexecution, potentially shared remote reads are replayed from logs, while potentially shared remote writes are squashed to avoid data pollution to forward-executing tasks. We skip logs for local reads/writes to reduce error-free overhead, and to use remote write logs to bring memory state up to date, so local reads/writes can be simply reexecuted. Collective communications are treated as combinations of read/write operations, and log/reexecute accordingly.

Remote reads do not need to be ordered with respect to local reads/writes, because they will never touch overlapping regions. The interleaving between incoming remote writes and local reads/wirtes, however, needs to be preserved to correctly reexecute them in an order consistent with that observed by non-reexecuting tasks. Each task maintains two counters, *Epoch Counter*

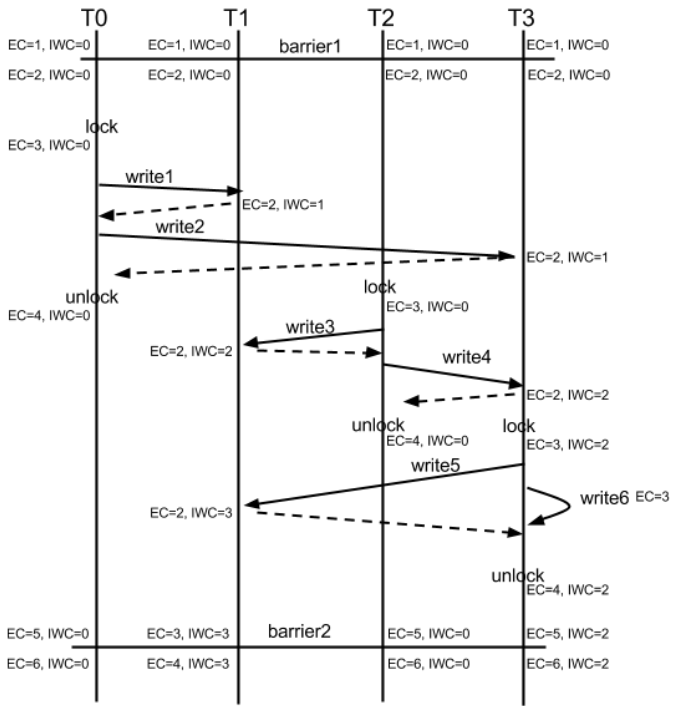

Figure 3: Example for EC and IWC.

(EC) and *Incoming Write Counter (IWC)*, to preserve the interleaving pattern. EC is maintained by software and is incremented at every synchronization event (i.e., locks, fences, barriers, and strict reads/writes). IWC, on the other hand, is managed by write-target-side NIC hardware and is incremented each time a NIC sees an incoming write operation. Two other implicit counters are also used to restore the interleaving pattern: *Program Order(PO)* and *Log Sequence Number (LSN)*. Program order is the executing order of instructions in computer code, while log sequence number is monotonically incremented by each task each time a log entry is generated.

Remote writes are logged in three steps: (1) when a task initiates a remote write, it generates one write log entry at the write-initiator side to log the accessed address and write value; (2) the write goes through the network, and the ACK message carries the write-target side EC and IWC to the write initiator; and (3) the write initiator then generates another log entry to log the target-side EC and IWC values. This way, all remote writes will have target-side EC/IWC associated with them.

When a failure happens, the failed CD gathers write logs from its relaxed group, rewinds the EC to *EC_begin* (EC value at CD begin), and then starts reexecution. In reexecution, the EC is incremented the same way with forward execution, and memory state is restored epoch by epoch. In each epoch, first, all incoming write logs, whose EC is the same value with the current epoch, are replayed in increasing IWC order. Then, local reads/writes are reexecuted and remote reads are replayed when they are reexecuting.

In each epoch, we arbitrarily choose to replay incoming remote writes first, and reexecute local reads/writes second. This order is reasonable based on the assumption that applications are properly synchronized. Hence in the same epoch, incoming writes and local reads/writes will not touch overlapping regions. UPC/UPC++, however, allows racy codes as long as every task views the race order the same way. Current PGAS CDs cannot guarantee to precisely restore the race order to match that of forward execution.

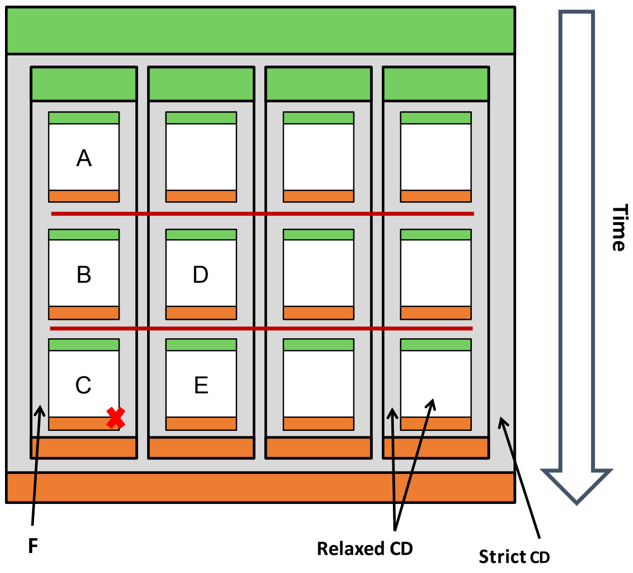

Figure 4: Log Management Example.

Besides communications and synchronizations, non-deterministic runtime events need to be logged as well. For example, random number generation functions need to be logged to regenerate the same random number in reexecution. Memory allocation should also be logged because other CDs may access allocated memory regions and failed CDs should provide the same allocated address in reexecution. Memory deallocation is deferred to the completion of a common ancestor CD that spans memory allocation and deallocation pairs.

Figure 3 shows a similar but more complex example as Figure **??**. This example shows that remote writes can be correctly reordered with the support of the EC and IWC. Note that at each synchronization event, such as lock/unlock operations, the initiator-side EC is incremented once to bring local execution into the next epoch. Barriers, on the other hand, increment the EC twice for each task: once at the entrance of the barrier and once at the exit to ensure all tasks enter into new epochs. In this example, *write4* and *write6* are ordered by EC of *T3*; *write1*, *write3* and *write5* are order by IWC in *T1*, and *write2* and *write4* are ordered by IWC in *T3*.

Log Management

CDs create read/write *log stores* when a CD object is created, and generate/replay logs to/from these log stores when CDs are in execution/reexecution. However, it is not safe to delete communication logs when a relaxed CD completes, because a relaxed CD may be reexecuted even after it completes because of escalation. Figure 4 provides an example of such a situation. In this example, one top level strict CD has 4 parallel relaxed child CDs (e.g., CD F), and each relaxed child CD has 3 sequential relaxed CDs (e.g., CD F has relaxed children CD A, B, and C). The red lines in this example represent global barriers across all tasks. Failures in CD C may be escalated to CD F, and CD F will reexecute CD A, B, and C sequentially. CD A and CD B, in this case, need their communication logs to provide consistent recovery.

Instead of deleting communication logs at a CD's complete time, relaxed CDs should push their communication logs to their parent CDs. Safe points to delete communication logs are completions of strict CDs, because by definition all communications are contained within one strict CD.

Thus, CDs accumulate logs during forward execution. These logs may grow rapidly and exceed a reasonable size before a strict CD is reached and logs can be safely deleted. The logs are periodically pushed out to more reliable media, like a local filesystem or PFS, and the space used for logs is garbage collected to reduce the memory footprint of the communication logs.

Node-level Dependency Tracking

With the logging mechanisms, relaxed CDs can recover by themselves without the need to propagate recovery to other sibling CDs. However, for some rare cases like memory data corruption or loss of communication logs, the failures are escalated to the least common strict ancestor CD to recover. Even though this escalation may be rare, falling back to a strict CD may be quite expensive, especially for some communication patterns. For example, for the nearest neighbor communication pattern, a common HPC communication pattern, falling back to strict CDs means global recovery, which is extremely expensive.

To tackle this problem, we implement a coarse-grained dependency tracking mechanism on top of communication logging to limit recovery propagation in such escalation cases. Since such escalation cases are rare, the dependency tracking mechanism only tracks coarse-grained dependencies (node-level) to reduce the overhead during error-free execution.

Before discussing the design details, we define the criteria identifying a dependence between a CD to a failing CD:

1. Any CD that is on the same node with the failed CDs.
2. Any CD that remotely write to the node where the failed CDs reside in.
3. Any CD on a node that the failed CDs remotely read.
4. Any CD that remotely writes to the node where the failed CDs remotely read from.

In escalation, instead of reexecuting the whole relaxed group, only CDs that a failed CD depend on will reexecute with the failed CD. These CDs, that a failed CD depend on, do not need to propagate the recovery to the CDs that they depend on because they do not suffer failures that require escalation, and PGAS CD semantics guarantee they can recover in isolation. For example, for nearest-neighbor communication, only neighbors of failed CDs will be reexecuted with failed CDs, instead of triggering global coordinated recovery.

To implement coarse-grained dependency tracking, my CD runtime maintains two lists for each node: a *read list* to track targets for remote read requests, and a *write list* to track initiators for incoming remote writes. The CD runtime either associates the two lists with a node-level CD if a node-level CD exists, or associates the two lists with the lowest CD level that is coarser than a node. When a CD on node M remotely writes to node N, the runtime records node M in the *write list* of node N, while when a CD on node M remotely reads from node N, the runtime records node M in the *read list* of node N. We do not expect the two lists to be large for common regular communication patterns,[5] because these communication patterns mainly exchange data with a limited number of tasks.

In cases of escalation, the CD runtime first pushes all communication logs of non-failed CDs to reliable media to avoid further log loss, and then recovers: (1) all CDs on the same node with the failed CD; (2) all CDs that exist on nodes in the write list; (3) all CDs that exist on the nodes in the read list; and (4) for each node in the read list, all CDs that exist on node in their write list. Note that (4) propagates recovery one more time to recover all CDs that failed CDs depend on.

Figure 5 gives an example of how dependency tracking can limit recovery propagation. Here nearest neighbor communication is used as an example. Each CD reads from its neighbor CDs (depicted with green arrows) and then writes to local data. This is the communication pattern in many HPC applications, like LULESH [4] and HPGMG [5]. All CDs' read list contains only neighbor CDs while their write lists are empty. When CD B fails and triggers escalation, it will trigger recovery of CD A and CD C, but all other CDs can continue their forward execution. Without dependency tracking, all CDs have to recover to obey CD semantics.

[5]Communication targets do not depend on input data values.

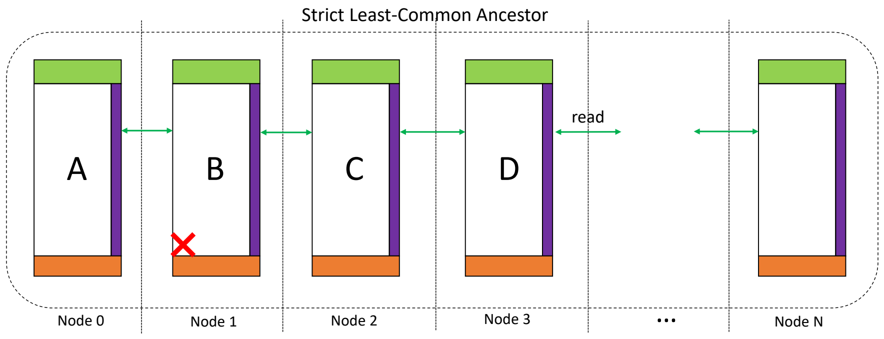

Figure 5: Dependency Tracking Example.

It is worth noting that synchronizations do not contribute to dependency lists, because synchronizations are not "actual" communications. We already have event logs to handle synchronizations for relaxed CDs (logged in forward execution and skipped in reexecution).

1 Bibliography and References

References

[1] M. Besta and T. Hoefler. Fault tolerance for remote memory access programming models. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 37–48, New York, NY, USA, 2014. ACM.

[2] CD Team. Containment Domains API. http://lph.ece.utexas.edu/users/CDAPI, 2017.

[3] J. Chung, I. Lee, M. Sullivan, J. Ryoo, D. Kim, D. Yoon, L. Kaplan, and M. Erez. Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems. In *the Proceedings of SC12: the ACM/IEEE International Conference on High-Performance Computing, Networking, Storage, and Analysis*, pages 58:1–11, Salt Lake City, UT, November 2012.

[4] I. Karlin, J. Keasler, and R. Neely. Lulesh 2.0 updates and changes. Technical Report LLNL-TR-641973, Lawrence Livermore National Laboratory, August 2013.

[5] H. Shan, S. Williams, Y. Zheng, A. Kamil, and K. Yelick. Implementing High-Performance Geometric Multigrid Solver with Naturally Grained Messages. In *2015 9th International Conference on Partitioned Global Address Space Programming Models*, pages 38–46, Sept 2015.