MODC: Resilience for disaggregated memory architectures using task-based programming

Kimberly Keeton[1], *Sharad Singhal, Haris Volos*[2]*, Yupu Zhang*[3]*,*
Ramesh Chandra Chaurasiya, Clarete Riana Crasta, Sherin T George,
Nagaraju K N, Mashood Abdulla K, Kavitha Natarajan, Porno Shome, Sanish Suresh

Hewlett Packard Enterprise

Abstract

Disaggregated memory architectures provide benefits to applications beyond traditional scale out environments, such as independent scaling of compute and memory resources. They also provide an independent failure model, where computations or the compute nodes they run on may fail independently of the disaggregated memory; thus, data that's resident in the disaggregated memory is unaffected by the compute failure. Blind application of traditional techniques for resilience (e.g., checkpoints or data replication) does not take advantage of these architectures. To demonstrate the potential benefit of these architectures for resilience, we develop *Memory-Oriented Distributed Computing (MODC)*, a framework for programming disaggregated architectures that borrows and adapts ideas from task-based programming models, concurrent programming techniques, and lock-free data structures. This framework includes a task-based application programming model and a runtime system that provides scheduling, coordination, and fault tolerance mechanisms. We present highlights of our MODC prototype and experimental results demonstrating that MODC-style resilience outperforms a checkpoint-based approach in the face of failures.

1 Introduction

Recent technology advances in high-density, byte-addressable non-volatile memory (NVM) (*e.g.*, [23,37,40,43,46]) and low-latency interconnects (*e.g.*, [2,8,29]) have enabled building rack-scale systems with a large disaggregated memory pool shared across decentralized compute nodes.

Disaggregated memory architectures present independent failure domains and a partial failure model: computations or the compute nodes they run on may die, but disaggregated memory remains available. When a compute node fails, updates propagated to disaggregated memory remain visible to other compute nodes. This partial failure model avoids a complete system shutdown in the event of a component failure. This failure model is an improvement over the failure model of scale up architectures, where a compute failure causes the failure of the entire application (and potentially the failure of the entire machine). It's also an improvement over the failure

model of scale out architectures, where a compute node failure may render a part of the dataset unavailable, unless the data is stored redundantly across nodes.

The applications that thrive in a disaggregated memory environment have large and long-lived datasets that aren't amenable to traditional resilience techniques such as checkpointing. For example, large-scale graph analytics applications at social media companies like Facebook operate over graphs with billions of vertices and trillions of edges that consume hundreds of petabytes [14]. Such applications may benefit from maintaining data in disaggregated memory for resilient computation on the graph data.

We develop *Memory-Oriented Distributed Computing (MODC)*, a framework for programming disaggregated architectures, by borrowing and adapting ideas from task-based programming models, concurrent programming techniques, and lock-free data structures. Task-based programming achieves fine-grained recoverability without explicit checkpoints. Developers write applications as a collection of idempotent tasks that operate on data items resident in disaggregated memory, and specify data and control dependencies between the tasks. The underlying runtime system uses this dependency information to schedule the tasks such that dependencies are satisfied, and to restart tasks if they fail.

The programming model is layered on top of a runtime system designed for a disaggregated environment where the disaggregated memory may not have general-purpose computational capabilities. As a result, we do not use server processes to actively manage scheduling, failure detection and coordination. Instead, our approach leverages shared data structures in disaggregated memory to encapsulate global state. Workers in a user application access these concurrent data structures in a decentralized fashion through a library interface to the MODC runtime. The runtime library uses one-sided operations and lock-free programming techniques to operate on the shared data structures, which ensures deadlock freedom and progress of the runtime in the event of compute failures. In particular, disaggregated memory holds dependency-tracking metadata and per-worker work queues, which enables work stealing to respond to failures and load imbalances.

We demonstrate that MODC enables resilient applications with lightweight failure recovery (*e.g.*, with less than 1% penalty, as compared to failure-free execution). A checkpoint-based failure recovery approach is as much as 51% slower than the MODC approach.

Authors' current affiliations:
[1]Unaffiliated, [2]University of Cyprus, [3]Google

2 Background

We begin by providing background on task-parallel programming models and disaggregated memory systems, as well as stating our system model.

2.1 Task-parallel programming

In task-parallel programming [13], the programmer decomposes an application into units of work, called *tasks*, that may execute in parallel. Programmers focus on structuring their program to expose parallelism, while an underlying runtime system takes on the responsibility of dynamically scheduling and executing tasks among a pool of workers.

Traditionally, tasking systems have primarily relied on global checkpoint/restart mechanisms for tolerating task failures. However, recent work has investigated other approaches to handling task failures. Resilient X10 [19] proposes extensions to the X10 task-parallel language [17] to expose failures to programmers, who can then handle individual task failures by exploiting domain-specific knowledge. CIEL [36] and Ray [34], like us, provide transparent lineage-based fault tolerance. They log function calls and require arguments and results of tasks to be immutable, so that they can use the lineage to transparently restart individual tasks on the event of a failure. A key difference is that our work leverages disaggregated memory to fail over computation from a failed worker to another live worker in a decentralized manner, by letting live workers steal work from a failed worker.

Dataflow systems, such as MapReduce [20], Dryad [25], and Spark [47] adopt a bulk-synchronous parallel (BSP) execution model, where all tasks within the same stage perform the same computation and cannot recursively create other tasks. MapReduce limits the dataflow to a bipartite graph comprising map and reduce tasks. Dryad extends the MapReduce model to allow computation to be expressed into a more general directed acyclic graph (DAG), but lacks support for dynamic task graphs, which can be useful for iterative and incremental computation. Spark does support dynamic task graphs, by dynamically tracking task dependencies through a centralized scheduler and adopting lineage-based fault tolerance. However, invoking the centralized scheduler at the end of each task adds overhead and increases latency. Streaming dataflow systems, such as Naiad [35] and Flink [15], implement the dataflow using a graph of continuous operators or long running tasks, avoiding frequently invoking a centralized scheduler, but limiting support to static task graphs.

Architectures based on dataflow languages [24] control program execution using *actors* that consume *objects* as inputs and produce objects as outputs. While these architectures also track dependencies among actors as part of program execution, the underlying actors are much more granular than our tasks, rely on specialized hardware for program execution and dependency management, and are not focused on resilience.

2.2 Rack-scale disaggregated memory

Resource disaggregation is a popular approach for specializing resources allocated to a workload. Increasing attention is being paid to leveraging memory resources outside of traditional compute nodes. Here we focus on *disaggregated memory*, rather than remote memory solutions where compute nodes share their memory with other compute nodes. In disaggregated memory system organizations, memory that is distinct from compute node memory is exposed over a fast network. Disaggregated memory may be constructed with intelligent memory servers containing general-purpose processors (*e.g.*, [31,32]) connected via RDMA (*e.g.*, [11,33,42]) or special purpose interconnects (*e.g.*, [38,41]), or as network-attached memory controllers with more limited intelligence (*e.g.*, [3, 10]). Depending on the implementation, disaggregated memory may also be augmented with additional support for metadata operations (*e.g.*, [44]). Disaggregation provides separate fault domains between processing and memory, meaning that the failure of a compute node doesn't render disaggregated memory unavailable.

2.3 Target environment and assumptions

In this work, we target a disaggregated memory architecture containing a high-capacity pool of memory that can be shared by compute nodes at low latency. This memory is directly connected to the interconnect, as defined in the Gen-Z standard [21], without an explicit memory server with general-purpose processing capabilities. Memory controllers serve non-volatile memory (NVM) on the fabric, which offers durability without the overheads of traditional storage devices. Compute nodes also have local memory, which is treated as private, while the disaggregated memory is treated as shared.

Compute nodes access disaggregated memory using load and store operations, atomic memory fabric operations (*e.g.*, compare-and-swap (cas)) and one-sided (*e.g.*, get and put) operations. Disaggregated memory presents a *logical address space* that is uniform across all compute nodes. Thus, any compute node can access any part of disaggregated memory in a byte-addressable manner.

Since rack-scale disaggregated NVM forms the primary persistence layer, it needs to be protected against memory hardware failures to avoid data loss. Hardware redundancy schemes have been proposed to protect both DRAM (*e.g.*, [28,48]) and NVM (*e.g.*, [22,26,30]). While these schemes protect against memory cell or chip failures, they do not protect against failures that render a whole memory node unavailable, such as memory controller or power supply failures. Hence, we assume a hardware redundancy scheme is used in combination with software-managed memory replication to protect disaggregated NVM. Using memory-side accelerators (e.g., [27]) can decentralize and minimize compute node overheads for memory replication. Additionally, we assume that

the memory interconnect incorporates sufficient component and path redundancy to prevent partitions.

3 MODC programming model

Programmers use resilient parallel jobs and tasks as a way to decompose and recover data processing computations. A resilient parallel application is structured as a set of restartable execution units (idempotent *tasks*). Each task is defined as a function with an explicitly specified set of disaggregated memory-resident inputs and outputs. It applies a transformation on its input data to produce its output data. The output of one task may serve as an input to another task, creating an input-output *data dependency*. Inputs and outputs are specified by names, which are resolved at runtime, thus permitting decoupled execution of producer and consumer tasks. A producer task can produce output data without knowing which consumer task (if any) will consume it, and a consumer task can consume data without knowing which task produced it. Similarly, output data may be treated as a future, allowing a consumer function to be scheduled before its inputs are fully generated; its execution is delayed until those inputs are ready (i.e., when the task(s) that produce the data are done).

Additional *control dependencies* (e.g., completion of all tasks in a set or task ordering that depends on the specific values produced rather than the existence of data) are expressed using a *job* abstraction, which describes a collection of tasks. Control dependencies may exist between jobs, but not between tasks within a single job. An application is comprised of one or more jobs and their constituent tasks (Figure 1).

Expressing data dependencies with named data items rather than explicit communication between tasks means that tasks don't need to be aware of where they (or their predecessors or successors) are run. Since tasks can be scheduled for execution on an arbitrary node, and may be restarted on a different node in case of failure, they are required to be *idempotent*: given the same input, they are expected to produce the same output. To enable restarting an idempotent task, the task's input data must remain available until the task successfully completes. Tasks are also expected to be *side-effect free* in the sense that they don't have side channels for communicating state that aren't visible to the framework.

The programming model supports a dynamic parallelism form of autoscaling, where work can be dynamically decomposed to provide (arbitrary) levels of parallelism, depending on the amount of data to be processed. The programmer defines the granularity of the jobs and tasks, and thus can strike a balance on a case-by-case basis between increased parallelism, lower chance of stragglers, and reduced recovery time from smaller tasks vs. increased efficiency from larger tasks (due to lower scheduling overhead and greater execution efficiency without intra-task communication).

The application dynamically spawns tasks, and manages control dependencies while the underlying framework handles
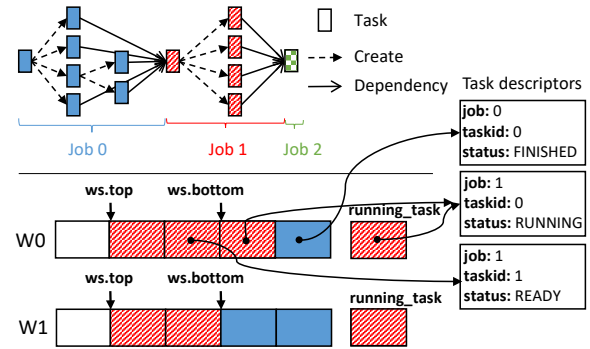

Figure 1: Resilient task execution by two workers.

data dependencies, task scheduling, and failure recovery. The framework asynchronously executes each task with all-or-nothing semantics, and re-schedules tasks that fail.

4 MODC runtime

MODC's runtime supports the programming model using a design that avoids any centralized service that may become a performance bottleneck or single point of failure. Space limits prevent us from describing the full runtime in detail; instead, we focus on the design principles and functionality most germane to scheduling and fault tolerance.

Design principles: MODC's runtime adheres to two design principles. *First, application and runtime global state is maintained as shared data structures in disaggregated memory that are visible to all participating processes, regardless of what nodes they run on.* Because all processes have direct access to global data structures, they can share data efficiently, service requests, and perform analyses for any part of a dataset, thus providing better load balancing and more robust performance in the face of skewed workloads. Shared access to global data also simplifies coordination, as participants need not exchange messages to establish a common view of global state. Processes maintain their persistent state in the shared pool, and since compute nodes and disaggregated memory fail independently, this persistent state survives the failure of any participating process or node. If a participant dies, any other participant can pick up where the failed participant left off, simplifying failure recovery. *Second, internally, the MODC runtime relies on lock-free synchronization for safe concurrent access to its internal data structures*. Lock-free synchronization typically involves splitting an operation into a non-critical section activity and a short critical section, which is accomplished atomically through a fabric-supported atomic such as a compare-and-swap (`cas`). Lock-free synchronization enables some active process to make progress despite other process delays or halting failures, offering us a powerful tool to handle compute node failures.

Design overview: MODC's scheduler is implicit, in that a parent task creates and schedules its child tasks. This approach

supports a dynamic parallelism model, where a logical task graph is created dynamically as new tasks are scheduled. The scheduling runtime functionality is carried out by *workers*, which track task dependencies, execute tasks whose dependencies are met, update scheduler metadata to report task completion, and re-execute failed tasks in the event of task, worker or compute node failures. The scheduling runtime itself is fault-tolerant, written using lock-free programming and concurrent, shared data structures in disaggregated memory.

The MODC framework assumes that a distributed application's parent job is instantiated by a workload manager that is external to the MODC runtime (e.g., mpirun() [6] or SLURM [9]). The workload manager instantiates a per-application worker pool of processes, where the initial pool size is determined by the application. The worker pool is per-application to reduce inter-application interference. The pool of workers may be dynamically resized as the application executes (e.g., if more than some threshold of the original workers fails, or if the average number of ready tasks per worker exceeds or falls below a threshold).

Each worker maintains its own task queue, which contains pointers to task descriptors for tasks that are ready to be executed (i.e., have all of their dependencies satisfied). Task queues are allocated in disaggregated memory, so that they remain accessible if a worker fails. An idle worker can steal tasks from another overloaded or failed worker. Work stealing enables dynamic load balancing and failing over a computation from a failed worker to another live worker in a decentralized manner without involving a centralized scheduler (which can become a bottleneck or single point of failure).

Enforcing control and data dependencies: Before a task can be added to a worker task queue, all of its control and data dependencies must be satisfied. The scheduling runtime tracks these dependencies using a collection of concurrent data structures in disaggregated memory. Internal scheduler metadata includes job and task descriptors containing metadata describing submitted jobs and tasks, and data structures to track progress on data and control dependencies.

To enforce control dependencies between tasks, the runtime must be able to identify in a fault tolerant manner when all tasks of a job complete execution. We take a work-oriented approach, where a group of workers executing tasks of a job reach consensus that they all finished executing tasks in their queues through the aid of a dynamic group barrier. When each worker finishes executing the tasks in its own queue, it tries to steal and execute tasks from other workers' queues. When it finds no more available tasks to steal, it waits on the barrier for all other workers to finish executing their tasks or until it receives a signal to steal work from a failed worker.

Like a traditional group barrier, our dynamic group barrier allows members of a group to collectively reach an execution point before they proceed. Additionally, it allows membership to change dynamically, so that active members don't wait for failed ones. The barrier comprises an array vector with a slot per member to indicate its participation in the barrier, two monotonically increasing sequence counters indicating membership changes and barrier releases, and a counter representing the number of members waiting on the barrier. All counters are placed contiguously in a 128-bit word so that they can be modified atomically using a 128-bit cas. When a failed worker is detected, a participant increments the membership sequence to indicate the change. When arriving at a barrier, a member counts the number of active members and increments the waiting counter; if the active member counter matches the waiting count, then it releases the barrier. Members spin-wait on the barrier by continuously reading the two sequence counters for changes either in membership or barrier release. Upon a membership change, each waiting member cancels its waiting, allowing it to resume and help with the execution of tasks previously assigned to the failed worker.

Executing tasks: Workers cooperatively execute tasks through work stealing, with each worker having its own task queue, which is structured as a lock-free circular-array deque [18]. Each worker pulls existing tasks from and pushes newly spawned tasks to one end of its queue, while other workers can steal tasks from the other end (Figure 1). Each queue descriptor has a field that identifies the queue's current owner worker process. The runtime represents each task through a task descriptor that holds scheduling-related information, including the associated job, the task ID within the job, the task's execution status, and the task's serialized closure. Each worker also maintains a slot with a pointer to the descriptor of its last running task. The queue, task descriptors and running slot are allocated in disaggregated memory, so that they remain accessible if a worker fails.

Task execution supports two stealing cases. First, idle workers can steal tasks from the top end of another worker's queue. If multiple workers attempt to execute the last task in a worker's queue, only one worker will succeed in issuing the cas to switch the task status from READY to RUNNING, thus guaranteeing exactly once execution. Second, upon a worker failure, a live worker can steal a task from the failed worker's running slot. A worker that finds a running slot with a READY task can simply take over and complete the stealing procedure, as described above. If the task's status is RUNNING (i.e., the worker failed before completing the task), then the runtime simply re-runs the task, since tasks are considered to be idempotent.

Detecting failed workers: The MODC runtime detects worker process failures in a decentralized manner, where each worker may detect the failure of another. Two global data structures provide the basic mechanism: a *global frontier counter* and a *worker vector* containing a heartbeat counter per worker to indicate the worker's notion of the current frontier. Each worker periodically reports its liveness status by incrementing its heartbeat counter. The global frontier counter is advanced when a sufficient number of workers reach the new value. Because the worker vector is stored in a designated

```
void pagerank(Matrix* graph, Vector* vec, float alpha, int iter, int total_iter) {
    if (iter != total_iter) {
        for (auto r = 0; r < graph.num_rows; r += L0_partition_size)
            createTask(spmv, graph, vec, alpha, iter, r, r + L0_partition_size);
        createJob(pagerank, graph, vec, alpha, iter+1, total_iter);
    }
}

void spmv(Matrix* graph, Vector* vec, float alpha, int iter, int rbegin, int rend) {
    // recursive decomposition
    if (rend - rbegin > L1_partition_size) {
        for (auto r = rbegin; r < rend; r += L1_partition_size)
            createTask(spmv, graph, vec, alpha, iter, r, r + L1_partition_size);
        return;
    }
    // sparse-matrix vector multiplication
    for (auto r = rbegin; r < rend; r++)
        vec[iter+1][r] = alpha * vec[iter][r] + (1-alpha) * graph->MultiplyRow(r, vec[iter]);
}
```

Figure 2: PageRank implementation using MODC tasks.

location in disaggregated memory that other processes can inspect, any process can periodically check the heartbeat status of another process. If an inspecting process observes that the target process' heartbeat counter is frozen for a configurable time, then it pronounces the target process dead and takes other corrective actions.

5 Case study: PageRank

We demonstrate the applicability of MODC's resilient task model to data analysis by implementing PageRank, a graph analysis algorithm that iteratively computes a rank for each vertex of a directed graph [39]. Figure 2 shows part of a PageRank application that uses recursive decomposition to parallelize the PageRank computation in two stages, where each job corresponds to a PageRank iteration.

First, the top-level *pagerank* function performs multiple iterations, where each iteration *iter* creates a fixed number of tasks; each of these tasks invokes the *spmv* function. *Pagerank* then creates a new job corresponding to the next iteration $iter + 1$, which contains a *pagerank* task that will generate the tasks for iteration $iter + 1$, as described above. Due to the control dependency between jobs, iteration $iter + 1$ won't begin until all of iteration *iter*'s tasks complete. Second, each *spmv* task performs sparse matrix vector (SpMV) multiplication over its assigned part of the adjacency matrix of the graph and the current PageRank vector to generate new rank values. Each *spmv* task may recursively (i.e., dynamically) decompose work and create new *spmv* tasks until a target partition size is reached.

5.1 Prototype and evaluation platform

Our evaluation goal is to understand the fault tolerance benefits of MODC, as well as its overheads. We have implemented a prototype of MODC's resilient task programming model and runtime system, including the components described in the previous section. Our implementation builds on several open source libraries, including lock-free data structures [7], memory management [4, 5] and fabric atomics [1].

Our emulation platform for disaggregated memory is an HPE Superdome X machine running Red Hat Enterprise Linux 7.2 with 288 cores and 12 TB DRAM. Each of the 16 NUMA sockets has an E7-8890 v3 CPU with 18 cores and 768 GB DRAM. We use Quartz [45] to emulate disaggregated memory performance. Quartz emulates compute nodes by binding each worker process to a NUMA socket, with processes evenly distributed across the sockets. It emulates disaggregated memory by binding a single instance of the `tmpfs` in-memory file system to a remote NUMA socket used exclusively for emulating disaggregated memory. This results in 768 GB of disaggregated memory with remote latency of 400ns, as measured using a memory latency-sensitive pointer-chasing microbenchmark. This latency is 3x slower than the local latency and on par with previous research proposals on rack-scale fabrics [38].

Our PageRank implementations use one-dimensional row partitioning (also known as vertex partitioning): each task is responsible for a set of rows and the non-zeroes within those rows. Each experiment uses a $64M \times 64M$ recursive matrix (RMAT) [16] graph, and results are averaged over four runs.

5.2 Sensitivity to task size

We begin our analysis by exploring the sensitivity of the MODC PageRank implementation to the task size. Figure 3 shows MODC execution time relative to the best execution time (empirically determined to be 15,000 rows). This experiment uses eight workers. MODC performs well for a wide range of task sizes: performance is within 5% of the best for tasks of 5000 rows to 100,000 rows, and within 17% of the best for tasks of up to 8M rows. We use tasks of 15,000 rows for our subsequent MODC experiments.
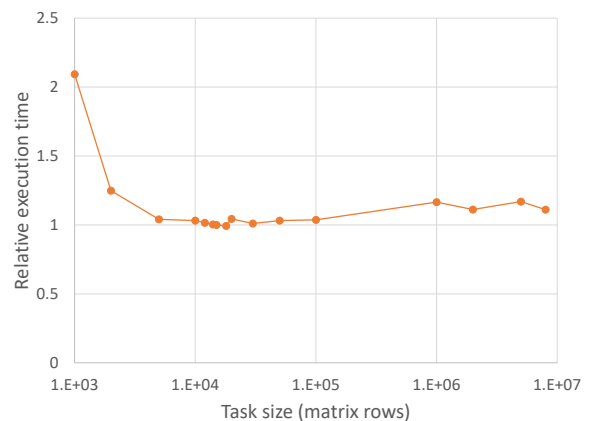

Figure 3: MODC PageRank sensitivity to task size

5.3 Fault tolerance

We compare the resilience of the MODC implementation of PageRank against the resilience of an MPI-based implementation with emulated checkpoints. In both cases, fourteen workers execute PageRank's iterative algorithm for 10 iterations. We inject a failure by crashing one of the workers, and let each implementation recover and continue execution. We assume non-shrinking recovery for these experiments: a new worker is instantiated upon a worker failure, to ensure that the number of active workers remains constant.

The MPI and MODC implementations use the same data structures and mmap data access methods, but use distinct scheduling and recovery mechanisms. The MODC implementation dynamically decomposes the problem into jobs and tasks. The failure is detected using MODC's heartbeat mechanism, and a new worker is activated from a hot standby spare. The task being executed when the worker died, along with any additional tasks in the failed worker's queue, are stolen for execution by one of the remaining workers.

The MPI implementation uses a bulk synchronous parallel programming model, with barriers between iterations. Each iteration's work is statically partitioned among the workers, with no work stealing. Partitioning assigns sets of 10,000 rows (the empirically determined best MPI set size) to workers in a round robin fashion. For fault tolerance, the MPI implementation uses checkpoints of the dense per-iteration output vector (512 MiB) to the emulated disaggregated memory. Recovery from a failure includes the time to read the checkpoint and the time to re-run all of the iterations since the last checkpoint. We assume a checkpoint interval of four iterations in the analysis below.
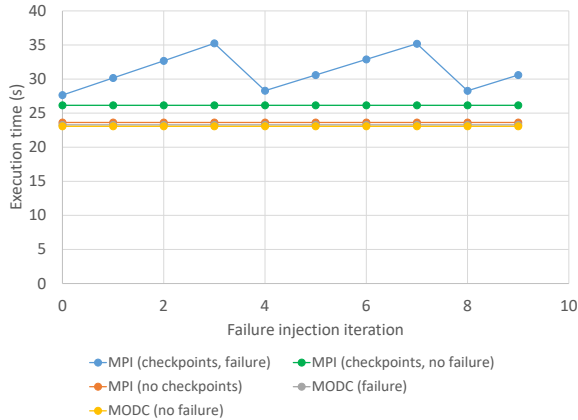

Figure 4: Comparison of fault tolerance approaches

Figure 4 illustrates the overall execution time for 10 PageRank iterations, with a failure injected at the iteration shown on the x-axis. We measure execution with and without the failure. With a failure injected, MODC only needs to restart the task that was executing when the worker crashed. As a result, its overall execution time with a failure is 23.3s, less than 1%

degradation from its failure-free execution. Execution time for MPI with checkpointing depends on how soon the crash occurs after the last checkpoint, ranging from 1.06X – 1.35X of the failure-free execution time. Checkpoint-recovered execution is slower than MODC resilient execution, ranging from 19% slower than MODC (if the crash occurs just after the checkpoint) to as much as 51% slower (if the crash occurs just before the next checkpoint).

6 Discussion

Although it is reasonable to ask whether it is realistic to require programmers to structure applications as a set of idempotent tasks, this requirement is a common one among execution frameworks that target fault tolerance (e.g., CIEL [36], Spark [47] and Ray [34]), as well as recent theoretical computational models [12]. We believe this model matches the needs of a large class of interesting parallel computations, including single program multiple data (SPMD), bulk synchronous parallel (BSP), and fork-join parallelism, as well as iterative and recursive algorithms.

The granularity of computation that makes sense as an idempotent task depends on the application's logic. As observed in Section 5.2, very small tasks incur noticeable dependency tracking overheads, while very large tasks provide limited ability to balance load through work stealing and may increase recovery time (due to large task restart). MODC successfully achieves near-optimal performance across a wide range of task sizes between these two extremes.

To address the concern of large task recovery times, we considered allowing applications to provide application-specific checkpointing, hidden from the MODC runtime. We found this approach inelegant, though, because it requires that the application-specific checkpoints are retained until the successful completion of the task, which requires additional runtime bookkeeping. Instead, we believe the cleanest way to avoid rework due to the restart of large tasks is for applications to use recursive decomposition to dynamically generate smaller tasks, and then let the runtime's dependency tracking handle executing and restarting the finer-grained tasks.

7 Conclusion

Disaggregated memory presents an opportunity to rethink how to support application resilience at rack scale. We proposed MODC, a framework that lets programmers write resilient applications using idempotent tasks. By placing the task dependency graph and work queues in disaggregated memory, MODC's runtime can fail over computation from a failed worker to a live worker in a decentralized manner. Our evaluation results show that MODC's fine-grained resilience can outperform traditional checkpoint-based approaches in the face of failures.

References

[1] Fabric-attached memory atomics library. https://github.com/FabricAttachedMemory/libfam-atomic.

[2] Gen-Z Consortium. http://www.genzconsortium.org.

[3] Hewlett Packard Enterprise The Machine. https://www.labs.hpe.com/the-machine.

[4] The librarian file system (lfs) suite for the machine. https://github.com/FabricAttachedMemory/tm-librarian.

[5] Non-volatile memory manager (nvmm). https://github.com/HewlettPackard/gull.

[6] OpenMPI mpirun(1) man page (version 4.0.0). https://www.open-mpi.org/doc/v4.0/man1/mpirun.1.php.

[7] Radix tree. https://github.com/HewlettPackard/meadowlark.

[8] RDMA Consortium. http://www.rdmaconsortium.org/.

[9] Slurm Workload Manager, version 18.08. https://slurm.schedmd.com.

[10] Marcos K. Aguilera, Kimberly Keeton, Stanko Novakovic, and Sharad Singhal. Designing far memory data structures: Think outside the box. In *HotOS'19*, pages 120–126, 2019.

[11] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Procedings of the VLDB Endowment*, 9(7):528–539, March 2016.

[12] G. Blelloch, P. Gibbons, Y. Gu, C. McGuffey, and J. Shun. The parallel persistent memory model. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[13] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[14] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*, pages 49–60, San Jose, CA, 2013. USENIX.

[15] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, August 2017.

[16] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SIAM International Conference on Data Mining*, 2004.

[17] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[18] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[19] David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, and Olivier Tardieu. Resilient x10: Efficient failure-aware programming. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, page 67–80, New York, NY, USA, 2014. Association for Computing Machinery.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design Implementation - Volume 6*, OSDI'04, page 10, USA, 2004. USENIX Association.

[21] Gen-Z Consortium. Gen-z core specification, version 1.1. https://genzconsortium.org/specification/gen-z-core-specification-1-1/, Accessed August 23, 2020.

[22] Oren Golov, Oren Segal, Uzi Doron, Julian Vlaiko, and Avraham Meir. Redundant storage in non-volatile memory by storing redundancy information in volatile memory, June 16 2015. US Patent 9,058,288.

[23] Hewlett Packard Enterprise. Persistent memory: The performance of memory with the persistence of storage. https://www.hpe.com/us/en/servers/persistent-memory.html.

[24] Ali R. Hurson. Dataflow computers: Their history and future. In Benjamin Wah, editor, *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2008.

[25] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev.*, 41(3):59–72, March 2007.

[26] R. Kateja, N. Beckmann, and G. R. Ganger. Tvarak: Software-managed hardware offload for redundancy in direct-access nvm storage. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 624–637, Los Alamitos, CA, USA, jun 2020. IEEE Computer Society.

[27] Daehyeok Kim, Amirsaman Memaripour, Anirudh Badam, Yibo Zhu, Hongqiang Harry Liu, Jitu Padhye, Shachar Raindel, Steven Swanson, Vyas Sekar, and Srinivasan Seshan. Hyperloop: Group-based nic-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 297–312, New York, NY, USA, 2018. Association for Computing Machinery.

[28] Jungrae Kim, Michael Sullivan, Sangkug Lym, and Mattan Erez. All-inclusive ECC: Thorough end-to-end protection for reliable computer memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 622–633, Piscataway, NJ, USA, 2016. IEEE Press.

[29] Patrick Knebel, Dan Berkram, Al Davis, Darel Emmot, Paolo Faraboschi, and Gary Gostin. Gen-z chipset for exascale fabrics. In *Hot Chips*, August 2019.

[30] Gregg B. Lesartre, Dale C. Morris, Gary Gostin, Russ W. Herrell, Andrew R. Wheeler, and Blaine D. Gaither. Fault tolerance for persistent main memory, December 31 2014. WO Patent App. PCT/US2013/048,759.

[31] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *International Symposium on Computer Architecture (ISCA)*, 2009.

[32] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch. System-level implications of disaggregated memory. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2012.

[33] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. Asymnvm: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *ASPLOS'20*, pages 757–773, 2020.

[34] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[35] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[36] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[37] Intel Newsroom. Intel and Micron produce breakthrough memory technology. http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technolo July 2015.

[38] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[39] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web, 1999.

[40] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[41] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *OSDI*, pages 69–87, 2018.

[42] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. Strom: smart remote memory. In *EuroSys'20*, pages 29:1–29:16, 2020.

[43] Dmitri B Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The missing memristor found. *Nature*, 453:80–83, May 2008.

[44] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *USENIX ATC 2020*, 2020.

[45] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A lightweight performance emulator for persistent memory software. In *Proc. of 16th Annual Middleware Conference*, Middleware 15, pages 37–49, New York, NY, USA, 2015. ACM.

[46] Yuan Xie. Modeling, architecture, and applications for emerging memory technologies. *IEEE Design & Test of Computers*, 28(1):44–51, 2011.

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28, 2012.

[48] Ruohuang Zheng and Michael C. Huang. Redundant memory array architecture for efficient selective protection. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 214–227, New York, NY, USA, 2017. ACM.