Portable Resilience with Kokkos

Jeffery Miles
Scalable Algorithms
Sandia National Laboratories
Albuquerque, NM, USA
jsmiles@sandia.gov

Nicolas Morales
Scalable Modeling & Analysis
Sandia National Laboratories
Livermore, CA, USA
nmmoral@sandia.gov

Carson Mould
Scalable Modeling & Analysis
Sandia National Laboratories
Livermore, CA, USA
cmould@sandia.gov

Keita Teranishi
Scalable Modeling & Analysis
Sandia National Laboratories
Livermore, CA, USA
knteran@sandia.gov

KEYWORDS

Checkpoint/restart, resilience, software redundancy, fault tolerance, software portability

ACM Reference format:

Jeffery Miles, Nicolas Morales, Carson Mould and Keita Teranishi. 2019. Performance Portable Resilience with Kokkos. In *Proceedings of ACM International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'19). ACM, New York, NY, USA, 2 pages.*

1 Poster Summary

Mission critical and scientific applications are challenged with being portable, executing efficiently on high performance and massively parallel hardware, and being tolerant to hardware faults. Historically portability has been the responsibility of the application developer, but with the introduction and advancement of Kokkos[1], a performance portable ecosystem, developers can focus more on their mission rather than the low-level details associated with targeting different hardware architectures. Kokkos achieves the goal of portable performance with abstractions and concepts that leverage modern C++ template meta-programming to enable configurable software that benefits from compile time optimizations. Fault tolerance in software has been widely studied and, in many cases, applied to mission critical and scientific software [2]. The challenge with fault tolerance implementations to date is the lack of consistency with regards to data structures and interface APIs. Much like having different hardware specific programming constructs, having different interfaces to each fault tolerance implementation makes portability difficult to achieve. By extending and expanding the abstractions and concepts within Kokkos, portable fault tolerance can be achieved in mission critical and scientific software with minimal effort from domain specific application programmers. The required concepts and abstractions have been extended within the Kokkos ecosystem to support a dynamically expandable set of fault tolerance methodologies and libraries. This summary and poster illustrate these abstractions and three specific additions which apply and validate the internals.

1.1 Kokkos Resilience Abstractions

The two high level concepts that are included with Kokkos resilience abstractions are Checkpoint/Restart and Software Redundancy. The goal with both additions is to be able to add fault tolerance to an existing Kokkos application with minimal end user changes, and subsequently simplify end user code for new Kokkos applications. The Checkpoint/Restart concept is implemented in two methods: manual and automatic. Manual uses the Kokkos view abstraction and the Kokkos memory space concept to give end users simple constructs to move application data from one context to another. Automatic checkpointing similarly uses the view abstraction, but instead of using the memory space concept, it provides a lambda style abstraction to capture data references for use in a checkpointing context. The checkpointing context then connects the end user data to the checkpointing interface hiding the underlying checkpoint API. Finally, Software redundancy is achieved by extending the Kokkos execution space concept with a redundant variation which encapsulates multiple instances of existing execution spaces and end user data.

1.2 Manual Checkpoint/Restart

Manual checkpoint/restart manages state data by mirroring end user views with checkpoint views. The checkpoint view is attached to a checkpoint memory space, which is different from typical memory spaces because the data is not expected to lie within directly accessible memory. The data in a checkpoint view may be in a network resource, a local file or an RDMA location. Checkpointed data is moved between the user view and the checkpoint views using deep copy operations, which internally use the underlying API required to access the remote resource. The checkpoint memory space internals also catalog and operate on the checkpoint mirror views, thus hiding the "work" required to execute a checkpoint and restart. As a result, the end user code to initiate a checkpoint (or restart) is reduced to a single line. To further simplify the end user code, the manual checkpoint subsystem also provides utilities to manage file system directories and attach directory names to checkpoint memory spaces. The manual checkpoint concept has been validated with HDF5 and std::streams backends using the "miniMD" app which is an existing MPI/Kokkos benchmark application.

1.2 Automatic Checkpoint/Restart

The term *automatic checkpointing* refers to checkpointing views without manual intervention by the user application. In Kokkos resilience, user code defines a checkpoint region by implementing a C++ lambda passed into the checkpoint function. The user data is accessed via Kokkos views, and the checkpoint region "captures" these views from the end user functor. During the capture process, the non-const view data are selected for checkpoint/restart. The advantage of this approach is that the implementation has very little impact on the logic or control flow of the end user application. By using C++ lambdas to abstract checkpoint regions, resilience logic stays within the Kokkos internals and not user code. In the case where Kokkos Resilience has detected that a restart is necessary, the lambda will not be executed and instead the checkpoint will be loaded. Stated differently, the checkpoint region should be treated as a black box in the sense that the code within the region may be executed, or the "resulting" data may be loaded from the checkpoint.

The automatic implementation raises some challenges that are easily overcome within Kokkos. In C++, it is typically impossible to introspect lambdas in order to detect what has been captured in the lambda. However, with Kokkos views as the accessor to the user data, this data can be detected and accessed through the copy constructor of the view. By enhancing the view copy constructor with a capture hook, Kokkos internals has the ability to process data within a functor using the hook. When the "enable hook" flag is enabled, the copied view data is detected and added to a checkpoint list. All of the user data captured in a lambda can then be accessed by Kokkos because when the lambda is copied, the views are also copied. The automatic checkpoint algorithm is illustrated below.

```
Precondition: list of views $Q$, $\lambda$ is a C++ lambda, $i$ is the current iteration
  function $checkpoint(\lambda, i)$
      ClearList($Q$)
      $c \leftarrow$ True
      $\lambda \leftarrow$ Copy($\lambda$)              ▷ calls the copy constructor for each view in $\lambda$
      $c \leftarrow$ False
  ▷ $Q$ now contains all views captured by $\lambda$
      if RestartRequired($i$) then
          for $q \in Q$ do
              LoadCheckpoint($q$)
          end for
      else
          $\lambda$ ()
          for $q \in Q$ do
              StoreCheckpoint($q$)
          end for
      end if
  end function
```

Figure 1: Automatic checkpointing algorithm

To verify the automatic checkpointing scheme, the restart detection, checkpointing, and restart operation have been implemented using the VeloC API[3] as a backend. This allows the application to take advantage of the performance benefits of VeloC using a nonintrusive C++ API.

1.3 Redundant Execution

The Redundant execution model is a direct extension of the execution space concept. A redundant execution space guarantees a higher level of services by duplicating input user data, replicating the execution, and then recombining the results. Similar to the automatic checkpointing, data duplication is achieved through Kokkos views that are captured from the end user functor. The duplicate execution spaces are launched concurrently using streams or tasks, and the recombination process performs a parallel "voting" exercise, where majority results are considered final and written to the end user views. In the backend implementation of the duplicate execution space, objects are instantiated such that the end user functor is copy constructed for each instance. This operation triggers the captured view data to be duplicated in the newly constructed objects, which reside on the target execution device. Once all of the concurrent execution objects have completed, a final parallel execution task is launched to recombine the modified views. The recombination process uses type specific operations for comparison. For enumerated types such and integer and long the operation is a simple comparison, but for single and double precision types the operation is an absolute difference. The redundant execution concept has been validated with the CUDA backend having streams enabled in the "miniMD" app, which is an existing Kokkos benchmark application that support the CUDA execution space.

2 Conclusions

Kokkos has achieved portability with software resilience by leveraging the abstractions and concepts within the application ecosystem. Checkpoint/Restart through Kokkos views reduces the extra work required from end user code to expose application data to a checkpointing subsystem. Having both automatic and manual implementations provides end users a choice of how to apply the checkpoint/restart concept to best fit the application structure and the resources available. The redundant execution concept is implemented using the existing Kokkos execution space concept with no additional API requirements, which in-turn allows applications to take advantage of this feature by changing a single line of end user code. Future advancements can then be managed internally within Kokkos, not requiring changes to the end user application. Finally, each these additional abstractions and concepts for software resilience can be applied to existing or new execution and memory space backends within Kokkos, which expand the capabilities moving forward, thus reducing the cost to design and operate mission critical and scientific software.

ACKNOWLEDGMENTS

Sandia National Laboratory is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

REFERENCES

[1] H. C. Edwards, C. R. Trott and D. Sunderland, (2014). "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," Journal of Parallel and Distributed Computing, 74 (12): 3202-3216. DOI: https://doi.org/10.1016/j.jpdc.2014.07.003
[2] I.P. Egwutuoha, D. Levy, B. Selic, S. Chen, (2013). "A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems", The Journal of Supercomputing, 65(3): 1302-1326. DOI: https://doi.org/10.1007/s11227-013-0884-0
[3] Argonne National Laboratory, "VeloC" Argonne National Laboratory, 2018. [Online]. Available: https://veloc.readthedocs.io/en/latest/.