SANDIA REPORT
SAND2019-3616
Printed Click to enter a date

Sandia National Laboratories

Software Resilience using Kokkos Ecosystem

Jeffery S. Miles
Keita Teranishi
Nicolas M. Morales
Christian R. Trott

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico
87185 and Livermore,
California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Rd
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods/

ABSTRACT

Due to the cost of hardware failures within mission critical and scientific applications, it is necessary for software to provide a mechanism to prevent or recover from interruptions. The Kokkos ecosystem is a programming environment that provides performance and portability to many applications that run on DOE supercomputers as well as smaller scale systems. These applications require a higher level of service due to the cost associated with each simulation or the critical nature of the mission. Software resilience enables an application of manage hardware failures reducing the cost of an interruption. Two different resilience methodologies have been added to the Kokkos ecosystem: checkpointing has been added for restart capabilities and a resilient execution model has been added to account for failures in compute devices. The design and implementation of each of these additions are described, and appropriate examples are included for end users.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

No table of figures entries found.

ACRONYMS AND DEFINITIONS

Abbreviation	Definition
HDF5	Hierarchical Data Format v.5
MPI	Message Passing Interface
RCES	Resilient Cuda Execution Space
RCMS	Resilient Cuda Memory Space

1. SOFTWARE RESILIENCE

Mission critical and scientific applications rely on systems that maintain state information and can autonomously handle failures due to environmental input and hardware faults. The benefits of adding resilience to these applications include reduced cost due to having to re-start failed simulations, increased interoperability by exposing state information, and increased level of service for mission critical applications. The choice of resilience implementation is determined based on the cost of a failure relative to the cost of resilience. For instance, in mission critical systems where failure can result in the loss of expensive equipment or human life, the cost of a failure is the highest possible. In scientific simulations that produce data for research, the cost of a detected failure is the expense associated with re-running the simulation; however, when the failures are not detected, the cost is even greater because of incorrect results leading to fault engineering designs. Applications that require state information to be passed to different stages or process also gain a benefit from a resilient architecture. The Kokkos ecosystem provides performance and portability to DOE applications through abstraction of parallel execution and memory concepts [1]. Through extensions to the parallel execution model and additions to the memory abstraction both execution and data resilience will expand the benefits of Kokkos implemented applications.

Two new concepts are added to the Kokkos ecosystem to enable application developers to quickly provide support for resilience with minimal change to existing or new Kokkos integrations. Failures can occur within hardware components that affect both the data in memory as well as the results of program execution. Resilience in program memory is accomplished through the concept of checkpointing. Checkpointing is defined as storing the internal state of application memory to a persistent media such that it can be recovered later if a failure is detected. The application can then restart from the last checkpoint and not loose hours of computations. Using existing parallel concepts, resilient program execution is accomplished by replicating a parallel code block such that the results are redundant. With the results from multiple replicas a program can choose the correct result by each against the others ignoring inconsistent contributions. The two resilience concepts above are accomplished within Kokkos by adding new spaces to already abstracted concepts. Checkpointing is enabled by adding a persistent memory space that accepts and receives data from the other Kokkos memory spaces. Likewise, resilient execution is accomplished by wrapping a resilient execution space around the parallel execution space.

The remainder of this document describes the details of resilient memory spaces and checkpointing as well as the resilient parallel execution model. Resilient Memory Space describes persistent memory spaces within Kokkos; Checkpointing with Kokkos provides examples of how to effectively add checkpointing to a Kokkos application; Resilient Execution Space provides details for the resilient execution space. Each section will describe the Kokkos implementation and provide examples demonstrating use. Note that the initial implementation will be limited to a small set of supported persistent memory spaces and one resilient execution space. These initial implementations are intended to demonstrate the concepts within Kokkos as well as test the modifications necessary to the Kokkos infrastructure.

1.1. Resilient Memory Space

The Kokkos data resilience concept starts with a Resilient Memory Space. The Resilient Memory space is an extension of the existing Kokkos memory space concept, where the data resides in a persistent memory architecture. Persistent data accessible through a Kokkos memory space allows applications to easily move data from volatile memory to persistent memory using Kokkos views and the deep copy function.

1.1.1. File System Memory Space

The simplest form of a persistent memory space is the local file system. A file system memory space enables an application to quickly move data from a Kokkos View to a local file and from the local file to a Kokkos View. The following example shows a code snippet illustrating the how to copy data from host space memory to the file system using Kokkos Views.

```
typedef Kokkos::LayoutLeft          Layout;
typedef Kokkos::default_memory_space  defaultMemSpace; // default
typedef Kokkos::FileMemSpace        fileSystemSpace; // file system

typedef Kokkos::View<double*, Layout, defaultMemSpace> local_view_type;
typedef Kokkos::View<double*, Layout, fileSystemSpace > file_view_type;

fileSystemSpace::set_default_path("path_to_data");
file_view_type F("file_name", N);
local_view_type A(N);
local_view_type::HostMirror h_A = Kokkos::create_mirror_view(A);

for ( ; ; ) {
  Kokkos::deep_copy(A, h_A);  // host to device
  Kokkos::parallel_for (N, [=](const size_t j) {
    A(j) = j;
   });
  Kokkos::deep_copy(h_A, A);  // device to host

  // Consistency check on result (h_A)
  if !consistency_check() {
    Kokkos::deep_copy(h_A, F);  // restore data and try again
  } else {
    Kokkos::deep_copy(F, h_A);  // save result and continue
   break;
  }
}
```

Figure 1-1. Simple Resilient Memory Space Example

In the Figure 1-1 the application data moves from the host space to the device space before the code block. Then the results are moved back to host space from the device space before eventually

being moved to the file system space. The data must go through the host space because the device space cannot access the file system. The name associated with the file system view is also used for the file stored on the local file system. For convenience the file system space uses a default path to store the files, but the name associated with the view and the file can also include path information. To make checkpointing more seamless the resilient memory space is further extended to include management hooks to connect data views with file system views automatically. The details of this concept are included in Checkpointing with Kokkos below.

1.1.2. HDF5 File Memory Space

Many scientific applications utilize the HDF5 file system for storing persistent data. HDF5 provides flexibility in the data layout within a stored file which may be necessary when storing large complex files. Using MPI, access to the file can also be parallelized using the HDF5 MPI I/O API. Using the HDF5 format also makes the persistent storage compatible with other applications that use the same format. This property is sometimes necessary for systems that move data between applications that are implemented in different architectures.

1.2. Checkpointing with Kokkos

In general application checkpointing involves writing multiple datasets to persistent storage at select stages of a computer simulation. This data can be used to restart the application in the event of a failure, but it is also useful for post processing operations. Regardless of the need, the checkpoint data must be store together with a little overhead as possible. The additions to the view and memory space concepts required to enable this type of checkpoint operation are as follows.

- A tracking mechanism to manage the file space views and the linked volatile memory views
- A mechanism to automatically create file space mirrors of host space views
- Functions to trigger the checkpoint and restore operations
- User provided function to check for consistency in order to determine system failure.

With these additions, the application designer can control the checkpoint operation for multiple dataset with minimal code additions. The code snippet in Figure 1-2 illustrates this approach.

Views are tracked by the label assigned when they are first created. This constraint requires that all checkpointed views have a unique name because the view label is also used to store the checkpoint file. All checkpointed files also use a file system space managed path, which is set using the file_system_space::set_default_path method. As seen in this example, the create_chkpt_mirror function connects the memory space view with the resilient space view. This operation is the hook that informs the underlying implementation which views are to be persisted. Note that the return type for this operation is the C++ concept "auto". The "auto" type is used for this command because the view type is constructed in-place with the C++ template defining the create_chkpt_mirror function. The create_chkpt_mirror function also accepts an override for the file path which is different from the file space default.

```
typedef Kokkos::LayoutLeft          Layout;
typedef Kokkos::DefaultExecutionSpace::memory_space
                              defaultMemSpace; // default device
typedef Kokkos::FileMemSpace        fileSystemSpace; // file system

fileSystemSpace fs;
typedef Kokkos::View<double*, Layout, defaultMemSpace> local_view_type;

local_view_type A("view_A", N);  local_view_type B("view_B", N);
local_view_type::HostMirror h_A = Kokkos::create_mirror_view(A);
local_view_type::HostMirror h_B = Kokkos::create_mirror_view(B);

auto F_A = Kokkos::create_chkpt_mirror(fs, h_A);
auto F_B = Kokkos::create_chkpt_mirror(fs, h_B);

fileSystemSpace::restore_all_views();  // restart from existing…
for ( ; ; ) {
  Kokkos::deep_copy(A, h_A);  Kokkos::deep_copy(B, h_B);
  Kokkos::parallel_for (N, KOKKOS_LAMBDA(const int j) {
    A(j) = j;  B(j) = j*2;
   });
  Kokkos::deep_copy(h_A, A);  Kokkos::deep_copy(h_B, B);

  if !consistency_check() {
    fileSystemSpace::restore_view("view_A"); // restore data
    fileSystemSpace::restore_view("view_B"); // restore data
  } else {
    fileSystemSpace::checkpoint_views();  // save result
  }
}
```

Figure 1-2. Illustration of Global Checkpointing

Also needed for the implementation seen above is a mechanism to check the consistency of the data in an application. Default consistency checking can only impose minimum constrains on the data stored in the view. Without knowledge of the final solution, testing the results of an operation directly is not feasible. Likewise, it may be too costly to verify the overall solution in shorter cycles. Thus, the consistency check applied in the context of data resilience is a reasonable solution to verify data integrity. For example, a view referencing a density field would expect that all values be positive non-zero numbers. Additionally, a view holding lookup indexes into another view would expect only values that are valid for the range of the reference view ranks. The checking operation is either a local function or a functor/lambda enabling a parallel execution pattern in order to maintain consistency.

1.2.1. Automatic Checkpointing

Automatic checkpointing is a process by which existing non-checkpointed code can adapt a checkpointing system without extensive overhead. The VeloC library from Argonne and Lawrence

Livermore National Labs provides an internal API to enable the checkpointing/restore process [2]; however, integrating his API is manual and not consistent with the Kokkos ecosystem. With automatic checkpointing VeloC functions to checkpoint data can be hidden from the user by capturing all the views that are used in a selected scope and providing them to the VeloC API. See Figure 1-3 below:

```
// Usage
Kokkos::View< ... > a( "a", ... );
Kokkos::View< ... > b( "b", ... );
Kokkos::View< ... > c( "c", ... );

for ( int iter = 0; iter < 100; ++iter )
{
  // Will generate "compute_stuff/<view>.<iter>.bin" for all captured views
  Kokkos::checkpoint( "compute_stuff", iter, true, KOKKOS_LAMBDA {
    Kokkos::parallel_for( N, KOKKOS_LAMBDA( int i ) {
      // Some computation with a and b
    } );

    Kokkos::parallel_for( N, KOKKOS_LAMBDA( int i ) {
      // Some other computation with a and c
    } );
  } );
}
```

Figure 1-3. Listing of the usage of Kokkos::checkpoint

The fundamental operation is the call to Kokkos::checkpoint, which takes a lambda (or functor) as the parameter. Similarly to Kokkos::parallel_for the checkpoint function introspects captured Kokkos views. Then each of these views are add to the checkpointed list using the VeloC API. In the example above, the files compute_stuff/a.<i>.bin, compute_stuff/b.<i>.bin, and compute_stuff/c.<i>.bin are generated for each iteration *i* after the lambda is executed. When VeloC detects a restart, each view is loaded from checkpointed data, and the lambda is not executed.

This model allows users to enable checkpointing making minimal changes to their code. Using a lambda provides a natural syntactical way of defining the scope, allowing users to group operations. Most use cases require checkpointing before and after a set of operations, so the grouping mechanism is imperative.

There are a few limitations to this particular model. Only views are checkpointed, so any variables that are necessary for computation should be wrapped in a Kokkos view. Additionally, similarly to Kokkos::parallel_for, checkpoint should not be called within a member function where this is implicitly captured because views may not be copied by value. Any view not captured by value will not be checkpointed.

1.3. Resilient Execution Space

Execution resilience is a parallel execution space where the policy is replicated thus creating redundant execution paths. A significant difference between the design of a resilient execution space and a resilient memory space is simplicity of the user additions required to enable resilient execution. A resilient execution space is pro-active redundancy meaning the implementation guarantees the integrity of the results. In the basic process flow of a resilient execution space, the resilient execution space spawns 3 equivalent internal execution spaces based on the requested policy. The 3 execution spaces then replicate the incoming data prior to execution and perform a consistency check on the final result by examining the resulting data from the 3 different execution paths. A very simple consistency check used in this design is a voting comparison where two or more results that are the same are taken as the final version. Below is a graphical illustration of this process.

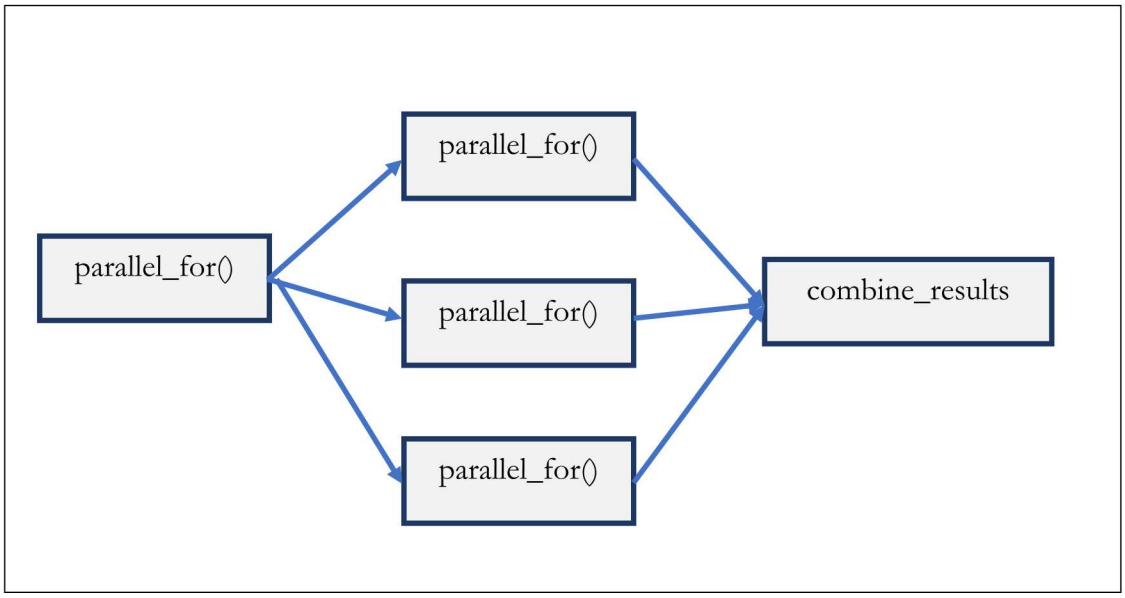

Figure 1-4. Graphical illustration of resilient execution space

The only requirement on the application to take advantage of the resilient execution model is to specify a resilient execution space in the outer policy. How the resilient execution space is specified has yet to be determined, but the method will pass this information to the parallel_for via the execution policy. The policy may be an execution space that wraps an existing parallel execution space, or the policy may be an existing policy with new parameters specifying resilient execution.

The implementation overhead for a resilient execution space is more significant than data resiliency because it involves capturing the views, managing the internal execution space and recombining the data. Capturing the views is similar to the process used in automatic checkpointing, but instead of linking two views together, duplicate view are created. Managing internal execution spaces involves constructing multiple parallel structures and executing each asynchronously. Finally, recombination of the view data requires a map of typed instances connecting the redundant views because of the comparison operation. The type information and pointers to the data are captured with the view and used after executing the policy during recombination.

The key aspect of implementing a resilient execution space is replicating views without significant interface changes in the Kokkos user context. Kokkos views can be implicitly managed by monitoring view copies when a capturing functor is copied. In most cases, the functor is copied into the ParallelFor structure which provides a manageable location to control the functor and subsequent view copies. The resilient version of the execution space simply aggregates multiple instances of the ParallelFor structure of the basic execution space. The resilient execution ParallelFor contains only a reference to the functor so that the only copies are those associated with the internal ParallelFor instances.

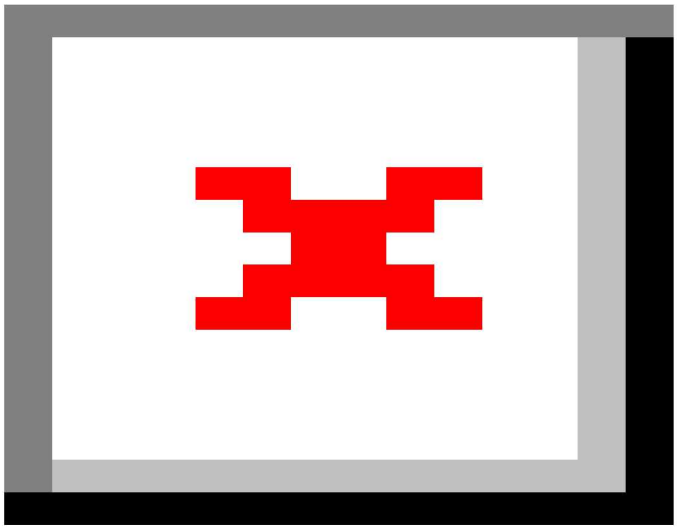

Figure 1-5. Resilient Cuda Execution Space

Figure 1-4 illustrates the Cuda Resilient Execution Space (CRES). The CRES works with the Resilient Cuda Memory Space (RCMS) to manage replicated views. When a const View is copied in the RCMS, the copy operates the same data as the CudaSpace. The view points to the original data

(managed via ref-counting) and only the mapping object is replicated. When a non-const View is copied in the RCMS, the copy creates a new memory allocation, copies the data from the original view, and creates an entry in a static map storing pointers to the data and objects containing the type information. This map of entries connecting duplicated views is maintained so that the data can be recombined and freed after executing the internal policy.

In the RCES, the ParallelFor creates 3 different instances of the Cuda Execution Space ParallelFor, which in-turn makes 3 different copies of the Functor and associated views. Note that the Copy of the non-const View on the device acts like a const View copy pointing to the duplicate's data because the device global memory must be copied from the host. After the 3 different ParallelFor closure methods are complete, the CRES ParallelFor uses functions in the RCMS to re-combine the duplicates back to the original views. After the non-const views are recombined successfully, the duplicate views are then freed, and the ParallelFor structures are cleared.

This same architecture can be applied to other execution spaces such as OpenMP or HPX; however, the value of the resilient execution may be limited if hardware isolation cannot be managed. Many OpenMP implementations for instance provide limited ability to selectively control at runtime which core each thread is launched. Without the ability to explicitly manage the threads' execution hardware, isolating thread blocks for redundant execution may be problematic.

REFERENCES

[1] H. C. Edwards, C. R. Trott and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing ,* vol. 74, no. 12, pp. 3202-3216, 2014.

[2] Argonne National Laboratory, "Veloc," Argonne National Laboratory, 2018. [Online]. Available: https://veloc.readthedocs.io/en/latest/.

DISTRIBUTION

Email—Internal

Name	Org.	Sandia Email Address
Technical Library	01177	libref@sandia.gov

This page left blank

This page left blank

Sandia National Laboratories

Sandia National Laboratories is a multimission 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.