Error Management

From Daxtoolkit

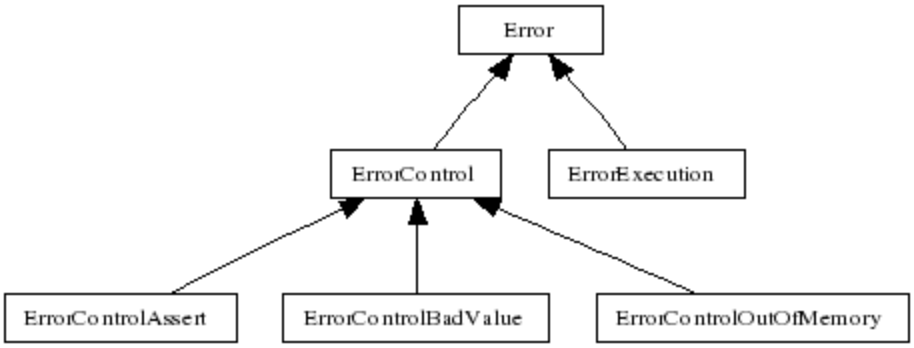
This document describes the error management mechanism in Dax. The basic error management is done through exceptions. Errors cause exceptions to be thrown where the can conveniently be caught by the user code. That said, exceptions cannot be simply thrown in the execution environment, so there is a special mechanism implemented to make it look as if an exception is thrown. There is also a similar, but slightly different, exception-based error management for checking conditions in tests.

Contents

Basic Exceptions in the Control Environment

In the control environment, an error is signaled by simply throwing an exception. Only classes that are subclasses of `dax::cont::Error` should be thrown from Dax. `dax::cont::Error` has the method `GetMessage` that can be used to print a description of what went wrong.

A `dax::cont::Error` is never created directly. Rather, one of its subclasses must be thrown. The current hierarchy of exception classes is as follows.



The class `ErrorExecution` is reserved for the special mechanism for reporting errors that happen in the execution environment (described below). `ErrorControl` is a superclass for possible errors that can occur in the control environment. Its subclasses are described in the immediately following two sections.

Raising Errors for Probable Problems

As with any production-quality API, Dax should be vigilant to ensure that any input it is given is properly formed. In cases where the input is malformed, Dax should quickly report a descriptive error (as opposed to blindly using garbage input and encountering some cryptic failure later). The type of error is identified by the subclass of `ErrorControl`, and each can provide a message giving context to the error.

ErrorControlBadValue

The `ErrorControlBadValue` class is thrown when a function or method is called with arguments that are invalid or inconsistent. As an example, the `ExecutionPackageField*` classes check to make sure that the size of the `ArrayHandle` passed to a worklet calling function matches the size of the field they will be mapped to. If the expected size is wrong, this error is thrown. The abbreviated code for that is here:

```
#include <dax/cont/ErrorControlBadValue.h>
...
class ExecutionPackageField
{
public:
  ExecutionPackageField(const dax::internal::DataArray<ValueType> &array,
                        dax::Id expectedSize)
    : Field(array) {
    if (array.GetNumberOfEntries() != expectedSize)
      {
      throw dax::cont::ErrorControlBadValue(
            "Received an array that is the wrong size "
            "for the field it is intended for.");
      }
  }
```

ErrorControlOutOfMemory

The operation of Dax requires it to allocated potentially large arrays, particularly in the execution environment. If this allocation fails, Dax should throw `ErrorControlOutOfMemory`. This could potentially be caught by the calling code, which might respond by calling Dax with some subdomain of the problem.

Here is an example of throwing `ErrorControlOutOfMemory` from `ArrayContainerExecutionCPU`. (The thrust container is pretty much identical.)

```
#include <dax/cont/ErrorControlOutOfMemory.h>
...
class ArrayContainerExecutionCPU
{
public:
  void Allocate(dax::Id numEntries) {
    try
      {
      this->DeviceArray.resize(numEntries);
      }
    catch (...)
      {
      throw dax::cont::ErrorControlOutOfMemory(
          "Failed to allocate execution array on CPU.");
      }
  }
```

Sanity Checks for Preconditions, Postconditions, etc.

In addition to checking user input for validity, it is good programming practice to consistently perform sanity checks to make sure that objects are in an expected state for valid operation. A good way to do this is to add "assert" statements that contain conditions that must be met for the operation to be valid. These asserts perform two very important functions. The first, and more obvious, function is to do a runtime check to make sure all necessary conditions are met and speed up debugging if a situation where there are not met is encountered. The second function is to document the expected state of an operation so that others reviewing the code will understand the proper operation.

Dax contains the header file `dax/cont/Assert.h`, which provides the macro `DAX_ASSERT_CONT`. This macro can be used in the exact same manner as the system `assert`, but if the assert fails the `ErrorControlAssert` class is thrown rather than crashing the program outright.

`DAX_ASSERT_CONT` should be used liberally to verify any expected conditions. For example, consider the `CopyFromControlToExecution` methods in the array containers. These are internal functions that should only be called with iterators that are valid and of the same size as the allocated array. The user should not be able to screw this up, but we should check anyway.

```
#include <dax/cont/Assert.h>
...
template<class T>
template<class IteratorType>
inline void ArrayContainerExecutionCPU<T>::CopyFromControlToExecution(
    const dax::cont::internal::IteratorContainer<IteratorType> &iterators)
```

```
{
  DAX_ASSERT_CONT(iterators.IsValid());
  DAX_ASSERT_CONT(iterators.GetNumberOfEntries()
                  == static_cast<dax::Id>(this->DeviceArray.size()));
  std::copy(iterators.GetBeginIterator(),
            iterators.GetEndIterator(),
            this->DeviceArray.begin());
}
```

Raising Errors in Worklets

Ideally, this exception-based error reporting would work in worklets and elsewhere in the execution environment. Unfortunately, that is not possible. Exceptions are simply not supported on Cuda devices. Even when exceptions are supported, such as in OpenMP threads, you cannot simply throw an exception and expect it to work. What happens when the exception reaches the invocation for the thread? The exception won't magically pass to the creating thread and consolidate itself with any other potential exceptions on other threads.

Nevertheless, Dax contains an error reporting mechanism for the execution environment that behaves very similarly to throwing exceptions. To report errors, each work object contains a method called `RaiseError`, which accepts a string describing the error. When this method is called on at least one thread, a `ErrorExecution` exception is thrown in the control environment where the worklet was scheduled.

As an example, let's say we have a simple worklet to take the square root of a scalar field. Furthermore, since the worklet will not handle complex numbers, we want it to raise an error if it receives a negative number. The implementation of this worklet could look something like this.

```
template<class CellType>
DAX_WORKLET void SquareRoot(
    DAX_IN dax::exec::WorkMapField<CellType> &work,
    DAX_IN const dax::exec::Field<dax::Scalar> &inField,
    DAX_OUT dax::exec::Field<dax::Scalar> &outField)
{
  dax::Scalar inValue = work.GetFieldValue(inField);
  if (inValue < 0)
    {
    work.RaiseError("Cannot take square root of negative number.");
    }
  dax::Scalar outValue = sqrt(inValue);
  work.SetFieldValue(outField, outValue);
}
```

As always, there will be a magic parser that builds some Dax control environment code to schedule the worklet. This scheduling function will handle the execution error and throw a `ErrorExecution` if one is encountered. For example, in the following code the catch block is run and prints out "Cannot take square root of negative number."

```
dax::Id arraySize = grid.GetNumberOfPoints();
std::vector<dax::Scalar> arrayBuffer(arraySize);
```

```
for (dax::Id index = 0; index < arraySize; index++)
  {
  arrayBuffer[index] = -2.5*index;
  }

dax::cont::ArrayHandle inArray(arrayBuffer.begin(), arrayBuffer.end());
dax::cont::ArrayHandle outArray(arraySize);

try
  {
  SquareRoot(grid, inArray, outArray);
  }
catch (dax::cont::ErrorExecution error)
  {
  std::cout << error.GetMessage() << std::endl;
  }
```

As you can see, calling `work.RaiseError()` is roughly equivalent to throwing an exception. The subtle difference, from the user's standpoint, is that the rest of the computation is undefined after the error is raised. Execution may stop immediately or execution may continue until completion. (For example, the `sqrt` function might still be called with a negative number.) The result array may be filled, it may be left alone, or may be replaced with garbage.

Sanity Checks in the Execution Environment

There is also a `DAX_ASSERT_EXEC()` macro in `dax/exec/Assert.h` that is roughly equivalent to the `DAX_ASSERT_CONT()` macro in the control environment. The only difference syntactically is that `DAX_ASSERT_EXEC` takes a work object for the second argument so that it can raise the error. Here might be a (rather pedantic) check for computing vector magnitude in a worklet.

```
template<class CellType>
DAX_WORKLET void Magnitude(
    DAX_IN dax::exec::WorkMapField<CellType> &work,
    DAX_IN const dax::exec::Field<dax::Vector3> &inField,
    DAX_OUT dax::exec::Field<dax::Scalar> &outField)
{
  dax::Vector3 inValue = work.GetFieldValue(inField);
  dax::Scalar magSquare = dax::dot(inValue, inValue);
  DAX_ASSERT_EXEC(magSquare >= 0, work);
  dax::Scalar outValue = sqrt(magSquare);
  work.SetFieldValue(outField, outValue);
}
```

You could, of course, use the assert to check other preconditions of the worklet. For example, you could shorten the implementation of the `SquareRoot` worklet with an assert, although that would result in a less readable error message.

```
template<class CellType>
DAX_WORKLET void SquareRoot(
    DAX_IN dax::exec::WorkMapField<CellType> &work,
    DAX_IN const dax::exec::Field<dax::Scalar> &inField,
    DAX_OUT dax::exec::Field<dax::Scalar> &outField)
```

```
{
  dax::Scalar inValue = work.GetFieldValue(inField);
  DAX_ASSERT_EXEC(inValue >= 0, work);
  dax::Scalar outValue = sqrt(inValue);
  work.SetFieldValue(outField, outValue);
}
```

Execution Error Implementation

Execution errors were implemented with changes to the work objects, the scheduling function in the DeviceAdapter, changes to the (soon-to-be-generated) worklet scheduling functions, and the introduction of the `dax::exec::internal::ErrorHandler` The abbreviated implementation of `ErrorHandler` is as follows.

```
class ErrorHandler
{
public:
  DAX_EXEC_EXPORT ErrorHandler(dax::internal::DataArray<char> m)
    : Message(m) { }

  DAX_EXEC_EXPORT void RaiseError(const char *message)
  {
    // Copy message into this->Message.
  }

  DAX_EXEC_EXPORT bool IsErrorRaised() const
  {
    return (this->Message.GetValue(0) != '\0');
  }

private:
  dax::internal::DataArray<char> Message;
};
```

`ErrorHandler` is a very simple object that holds a C-string array in a global memory space that is shared among all threads. A zero-length string (i.e. the first character is the null terminating character) indicates no error. Anything else indicates an error has occurred.

The work objects should be designed to hold an `ErrorHandler`. All the constructors (that are not copy constructors) should require an `ErrorHandler` being passed to avoid the odd situation of having to report an error of not having an `ErrorHandler`. The work object's `RaiseError` method simply delegates to the `ErrorHandler`.

```
template<>
class WorkMapField<dax::exec::CellVoxel>
{
  const dax::internal::StructureUniformGrid GridStructure;
  dax::exec::internal::ErrorHandler ErrorHandler;

public:
```

```
  DAX_EXEC_CONT_EXPORT WorkMapField(
      const dax::internal::StructureUniformGrid &gs,
      const dax::exec::internal::ErrorHandler &errorHandler)
    : GridStructure(gs), ErrorHandler(errorHandler) { }

  DAX_EXEC_EXPORT void RaiseError(const char *message)
  {
    this->ErrorHandler.RaiseError(message);
  }
```

Because the generation and checking of the C-string array can be device-specific, it is the responsibility of the DeviceAdapter to create and maintain it. The schedule method is responsible for creating an array on the device, setting it to a zero-length string before calling its functor, and returning the error string (or optionally null if there has been no error). As an example, here is the code that implements the thrust scheduling

```
namespace internal {

DAX_CONT_EXPORT
dax::thrust::cont::internal::ArrayContainerExecutionThrust<char> &
getScheduleThrustErrorArray()
{
  static dax::thrust::cont::internal::ArrayContainerExecutionThrust<char>
      ErrorArray;
  return ErrorArray;
}

}

template<class Functor, class Parameters>
DAX_CONT_EXPORT char *scheduleThrust(Functor functor,
                                     Parameters parameters,
                                     dax::Id numInstances)
{
  const dax::Id ERROR_ARRAY_SIZE = 1024;
  dax::thrust::cont::internal::ArrayContainerExecutionThrust<char> &errorArray
      = internal::getScheduleThrustErrorArray();
  errorArray.Allocate(ERROR_ARRAY_SIZE);
  *errorArray.GetBeginThrustIterator() = '\0';

  dax::thrust::exec::internal::kernel::ScheduleThrustKernel<Functor, Parameters>
      kernel(functor, parameters, errorArray);

  ::thrust::for_each(::thrust::make_counting_iterator<dax::Id>(0),
                     ::thrust::make_counting_iterator<dax::Id>(numInstances),
                     kernel);

  if (*errorArray.GetBeginThrustIterator() != '\0')
    {
    static char errorString[ERROR_ARRAY_SIZE];
    ::thrust::copy(errorArray.GetBeginThrustIterator(),
                   errorArray.GetEndThrustIterator(),
                   errorString);
    return errorString;
    }
```

```
  return NULL;
}
```

Although not shown here, the `ScheduleThrustKernel` is a simple functor that calls the provided functor with the parameters, the instance index, and the `ErrorHandler`.

The rest of the implementation deals with the code that schedules a particular worklet. On the execution environment, its functor simply builds a work object as normal but with an `ErrorHandler`.

> Note that this calling specification for the functor changed slightly. It now accepts a third parameter, which is the `ErrorHandler`. On a related note, the worklet scheduling functions had to be changed to construct the work in the kernel rather than on the host. This was so that it could be constructed with the `ErrorHandler` object, which was not available until running on the device. The change is probably a good idea anyway since the kernel needs to be able to modify the work object, and you don't want it shared. That problem has already caused the as-of-yet hardest bug I've had to track down. --Kenneth Moreland 11:13, 25 January 2012 (EST)

```
template<class CellType>
struct FooKernel {
  DAX_EXEC_EXPORT void operator()(
      FooParameters<CellType> &parameters,
      dax::Id index,
      const dax::exec::internal::ErrorHandler &errorHandler)
  {
    dax::exec::WorkMapField<CellType> work(parameters.grid, errorHandler);
    work.SetIndex(index);
    dax::worklet::Foo(work, parameters.arg1, ...);
  }
};
```

On the control side, the worklet schedule method has to capture the string that comes from the `Schedule` function in the DeviceAdapter and throw an exception if it is non-NULL and non-empty.

> My current implementation has the error simply returned as a string from the `Schedule`. I did it that way to minimize the amount of work done in the schedule function. However, perhaps it would be easier for everything if the schedule function threw the error itself. What does everyone else think? --Kenneth Moreland 11:13, 25 January 2012 (EST)

```
  char *error = DeviceAdapter::Schedule(
        dax::exec::kernel::FooKernel<CellType>(),
        parameters,
        fieldSize);

  if ((error != NULL) && (error[0] != '\0'))
    {
    throw dax::cont::ErrorExecution(error, "Foo");
    }
```

> Robert has been working on implementing the majority of these worklet scheduling functions in consolidated classes, one for each worklet type. I look forward to that. When such is the case, the

management of errors should go in there. --Kenneth Moreland 12:11, 25 January 2012 (EST)

Checking Conditions in Tests

As we write Dax, we have some momentum to write lots of tests. In particular, we have been writing "unit tests" that test a specific targeted functionality in isolation from all other functionality as much as possible. These functions are quick to write and quick to execute. They have proven to be very handy in catching problems with new functional units before plugging in to larger contexts (where problems become harder to find and fix). They also make for good regression tests, pinpointing exactly where something got broken.

As with any test program, these tests frequently tested conditions and reported errors when they were not as expected. To simplify our lives, we were creating helper functions like test_assert that performed the check and report in short and concise code. These functions were copied from one test to the next and slowly evolving. To simplify and shorten the code, I've consolidated this error reporting.

There is a new header file `dax/cont/internal/Testing` that defines several helpful facilities. First, it defines two macros to help detect and report test failures. The first is `DAX_TEST_ASSERT(condition, message)`, which takes the boolean expression `condition` and raises an error with `message` if it is false. The second is `DAX_TEST_FAIL(message)`, which always reports an error with `message`.

Both of these macros throw exceptions for errors that are expected to be caught, reported, and make the test return a failure condition. To simplify this second part, the header defines the function `dax::cont::internal::Testing::Run()`.

```
namespace dax {
namespace cont {
namespace internal {

struct Testing
{
public:
  template<class Func> static DAX_CONT_EXPORT int Run(Func function);
```

`Run` calls the given `function` with no arguments in a try block. It will catch test failures and report them correctly. It will also catch other Dax exceptions and others and report on those as well.

The following simple example comes from the `ArrayHandle` unit test. Notice how it uses DAX_TEST_ASSERT multiple times to check for conditions. Any errors are automatically picked up by the `Run` function. The `Run` function returns the integer value that should be returned from the test in general.

```
#include <dax/cont/ArrayHandle.h>

#include <dax/cont/internal/Testing.h>

namespace
{
const dax::Id ARRAY_SIZE = 10;
```

```
void TestArrayHandle()
{
  dax::Scalar array[ARRAY_SIZE];

  // Create an array handle.
  dax::cont::ArrayHandle<dax::Scalar>
      arrayHandle(&array[0], &array[ARRAY_SIZE]);

  DAX_TEST_ASSERT(arrayHandle.GetNumberOfEntries() == ARRAY_SIZE,
                  "ArrayHandle has wrong number of entries.");

  DAX_TEST_ASSERT(arrayHandle.IsControlArrayValid(),
                  "Control data not valid.");

  // Make sure that invalidating any copy will invalidate all copies.
  dax::cont::ArrayHandle<dax::Scalar> arrayHandleCopy;
  arrayHandleCopy = arrayHandle;
  arrayHandleCopy.InvalidateControlArray();
  DAX_TEST_ASSERT(!arrayHandle.IsControlArrayValid(),
                  "Invalidate did not propagate to copies.");
}


}


int UnitTestArrayHandle(int, char *[])
{
  return dax::cont::internal::Testing::Run(TestArrayHandle);
}
```

One problem with `dax/cont/internal/Testing.h` is that is adds a dependency to the control environment sources. Unfortunately, there are many unit tests for code in the base and execution namespaces that should not depend on control classes. Thus, there is a separate form of testing classes in `dax/internal/Testing.h` that does not rely on the control environment (by ignoring the #Basic Exceptions in the Control Environment). In this case, you would use the `dax::internal::Testing::Run()` function. The macros remain the same.

Acknowledgements

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.