# SANDIA REPORT

SAND2024-08882
Printed July 12, 2024

Sandia National Laboratories

# Sandia Toolkit Manual Version 5.21.1

Sandia Toolkit Development Team

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

**ABSTRACT**

This report provides documentation for the Sandia Toolkit (STK) modules. STK modules are intended to provide infrastructure that assists the development of computational engineering software such as finite-element analysis applications. STK includes modules for unstructured-mesh data structures, reading/writing mesh files, geometric proximity search, transfers, MPMD coupling support, and various other utilities. This document contains a chapter for each module, and each chapter contains overview descriptions and usage examples. Usage examples are primarily code listings which are generated from working test programs that are included in the STK code-base. A goal of this approach is to ensure that the usage examples will not fall out of date.

Note for Sandia classification reviewers: the source code for STK is open source, per Sandia SCR 1292.

This page intentionally left blank.

# CONTENTS

# LISTINGS

This page intentionally left blank.

# 1. STK OVERVIEW AND INTRODUCTION

The Sandia Toolkit (STK) modules provide infrastructure to support the development of computational engineering applications.

STK is composed of several modules:

- STK Util: various utilities such as parallel communication, command line parsing, etc.
- STK Topology: definitions of mesh-entity (elements, sides, etc) node orderings, side orderings.
- STK Mesh: parallel unstructured mesh
- STK IO: reading/writing of STK Mesh to/from Exodus files
- STK Coupling: support for MPMD coupling of MPI applications
- STK Search: geometric proximity bounding-box search
- STK Transfer: copy solution field values between meshes
- STK Balance: parallel load partitioning and dynamic rebalancing
- STK Middle Mesh: common refinement of two surface meshes
- STK SIMD: a general interface to vector instructions such as SSE, AVX, etc.
- STK ExprEval: string function expression evaluation

Some STK modules depend on others, but in general it is not necessary to use all of them together. For example, applications can use STK Search and STK Transfer without using STK Mesh. Also, STK Util doesn't depend on any other STK modules.

## 1.1. Building STK

STK contains cmake support, and can be fetched/built/installed using spack. Spack is the recommended way to build STK. STK is distributed within Trilinos, and thus leverages much of the cmake and spack support that the Trilinos ecosystem includes.

In general, building code libraries in Unix environments is complex and there are nearly endless variations and complications that might be encountered on specific platforms. Thus, here we will avoid giving too much detail and simply provide some general guidelines regarding dependencies, etc.

There is a subdirectory in the code-base, `stk/stk_integration_tests/cmake_install_test`, where you can find some examples that use cmake to configure and build subsets of the STK modules. Additionally, there are a couple of README files which contain "recipes" for using spack to obtain and build STK within Trilinos. Bear in mind that these all contain some machine-specific and user-specific nuances in terms of paths and dependency-versions, etc. But hopefully they can be useful in providing a starting point.

Notes regarding dependencies:

- Boost: STK used to have a mandatory dependence on Boost. This was removed in approximately the November 2022 timeframe, which corresponds to the 13.4.1 version of Trilinos. The spack package.py for Trilinos, as of spack version 0.21, imposed a mandatory dependence for stk on Boost. This was corrected for spack version 0.22 and now specifies that the dependence only exists through Trilinos version 14.0.0. What remains is an optional dependence on Boost for a stacktrace printing capability.
- Kokkos: STK modules depend on Kokkos. This will be discussed further in a later section; STK Mesh uses Kokkos for GPU support, as do some utilities in STK Util. STK has been tested and used with a number of Kokkos back ends, including Nvidia/Cuda and AMD/ROCm.
- MPI: Many STK modules use MPI parallelism. MPI is a mandatory dependency for STK Coupling, Search and Mesh, and anything that depends on any of those (such as IO, Balance, Transfer, etc).
- I/O: The STK IO module depends on the SEACAS/IOSS and SEACAS/Exodus libraries, which are also distributed with Trilinos. Those libraries in turn have dependencies on Netcdf, and optionally HDF5, and potentially others.
- Zoltan2: The STK Balance module depends on the Zoltan2 library, which is also distributed with Trilinos. That library in turn depends on libraries such as Parmetis and potentially others.

Spack provides a lot of help for the problem of finding and installing the "tree" of dependencies that is required for any non-trivial library build. Generally spack will issue an error if you attempt to specify a conflicting set of enabled and disabled dependencies. The STK configuration within Trilinos will automatically enable all STK submodules, but will disable any which have dependencies on libraries that have not been enabled. As an example, consider this minimal specification of trilinos:

```
spack add trilinos@15.0.0 +stk
```

Trilinos has MPI enabled by default, so many STK modules are enabled, but notably STK IO is not enabled (due to missing the exodus library) and STK Balance is not enabled (due to missing Zoltan2). Thus, to get a more complete STK installation, use this:

```
spack add trilinos@15.0.0 +exodus+zoltan2 +stk
```

Note that you also need to configure spack compilers, and add specs for appropriate infrastructure such as cmake and and an MPI implementation such as openmpi or others.

You may find that Kokkos is found and configured automatically by spack, but if you are doing a Cuda build (or other "non basic" builds) you probably have to configure Kokkos explicitly with the variants demanded by your particular system. For example, at least one of our local configurations requires something like this:

```
spack add kokkos@4.1.00+cuda+wrapper+cuda_constexpr+cuda_lambda \
       +cuda_relocatable_device_code~shared cuda_arch=70
spack add trilinos@master+cuda+cuda_rdc+exodus+stk+kokkos+wrapper \
       ~shared~boost cuda_arch=70 cxxstd=17
```

(The line-breaks in these commands are artificial, just for formatting.) Noteworthy about this configuration, is the disabling of shared libraries and the enabling of relocatable device code for Cuda. This is straying into specific platform aspects that are likely to fall out of date, but hopefully this provides a sampling of the variations that you may need, and a starting point for "google" searches to find more.

The "README" files mentioned above go through the whole process of setting up a spack environment, adding compilers and libraries, installing, and then using cmake and make to build a small test application that includes STK headers and links against STK libraries.

This page intentionally left blank.

# 2. STK UTIL

The STK Util module provides many utility capabilities that are used within STK modules and STK-based applications. The categories of utilities include error and exception handling, command-line argument processing, parallel operations, timing, string operations, etc.

Most of the documentation in this manual is in the form of code snippets demonstrating how to call STK functions and/or classes. These code snippets are pulled from 'live' test code, so they are guaranteed to be up-to-date at the time this manual is compiled. Many of the snippets pull in only part of the test file, leaving out some setup details, etc. Each snippet shows the name of the file it is from, so it is possible to go read the file to see all the details.

This first example is nearly the smallest possible STK test. It checks the number of MPI ranks and reports the STK version string and the `STK_VERSION` macro which are defined in the header `stk_util/Version.hpp`. Note that some STK modules may be built/run without depending on an MPI implementation. To facilitate this, STK provides some minimal MPI wrappers in the `stk_util/parallel/Parallel.hpp` header.

The STK version string takes the form of a tag/release version along with a git sha. If you need to request support from the STK development team, it is helpful to include this version string so that we know exactly which version/age of the code you have.

There is also a macro `STK_VERSION` which provides an integer version that can be used if API changes occurred in a particular version of STK, and you need your app to be able to build against multiple different versions. Thus you could use a comparison like `#if STK_VERSION > 5190200` (for example) if you wanted to use a STK symbol that didn't exist prior to that version. See the file `stk/CHANGELOG.md` for information about API changes.

**Listing 2.1 Example of retrieving the STK version string.**
**code/stk/stk_doc_tests/stk_util/VersionHowTo.cpp**

```
#include "gtest/gtest.h"
#include <stk_util/parallel/Parallel.hpp>
#include <stk_util/Version.hpp>
#include <iostream>
#include <string>

TEST(stkHowTo, reportVersion)
{
  if (stk::parallel_machine_size(MPI_COMM_WORLD) != 1) { GTEST_SKIP(); }

  const std::string stk_version = stk::version_string();
  std::cout << "This program is using STK version: " << stk_version << std::endl;
#ifdef STK_VERSION
  std::cout << "Value of STK_VERSION macro: " << STK_VERSION << std::endl;
#endif
}
```

Listing 2.2 demonstrates how to combine multiple reduction operations using the `stk::all_reduce` function, which uses `MPI_Allreduce` underneath.

**Listing 2.2 Example showing how to combine reduction operations**
**code/stk/stk_doc_tests/stk_util/AllreduceHowTo.cpp**

```
TEST(AllReduce, combinedOps)
{
  MPI_Comm comm = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(comm) != 2) { GTEST_SKIP(); }

  int myRank = stk::parallel_machine_rank(comm);

  constexpr int NumMin = 2;
  constexpr int NumMax = 3;
  constexpr int NumSum = 4;

  std::vector<int> ints(NumMin, myRank);
  std::vector<float> floats(NumMax, 1.0*myRank);
  std::vector<float> doubles(NumSum, 1.0*(myRank+1));

  stk::all_reduce(comm, stk::ReduceMin<NumMin>(ints.data())
                  & stk::ReduceMax<NumMax>(floats.data())
                  & stk::ReduceSum<NumSum>(doubles.data()));

  const int expectedMin = 0;
  const float expectedMax = 1.0;
  const double expectedSum = 3.0;

  for(int thisInt : ints) {
    EXPECT_EQ(expectedMin, thisInt);
  }

  for(float thisFloat : floats) {
    EXPECT_EQ(expectedMax, thisFloat);
  }

  for(double thisDouble : doubles) {
    EXPECT_EQ(expectedSum, thisDouble);
  }
}
```

Listing 2.3 shows an example of how to send a floating point value to all other processors. Note that there is a two phase process for packing the data into buffers. In the first phase, the data that is to be sent is used to *size* the communication buffer which will be sent to that processor. Then the `allocate_buffers()` call is made. Then, in the next phase, the same packing of buffers is done again, and the data is actually stored in the buffers. Finally, the `communicate()` call sends the buffers which are then unpacked by the receiving processors. In this example only one value is received from each processor.

The first snippet makes use of a helper function `pack_and_communicate` which is a convenience to hide the two phase approach and handles the calls to `allocate_buffers()` and `communicate()`. The second snippet performs the same operation using 'raw' `CommSparse` calls.

**Listing 2.3 Example showing how to communicate with other processors**
**code/stk/stk_doc_tests/stk_util/CommSparseHowTo.cpp**

```
TEST(ParallelComm, HowToCommunicateOneValue_PackAndCommunicate)
{
  MPI_Comm comm = MPI_COMM_WORLD;
  stk::CommSparse commSparse(comm);

  int myProcId = commSparse.parallel_rank();
  int numProcs = commSparse.parallel_size();

  double sendSomeNumber = 100-myProcId;

  stk::pack_and_communicate(commSparse, [&commSparse, &sendSomeNumber, &myProcId,
            &numProcs]() {
    for (int proc=0;proc<numProcs;proc++) {
      if ( proc != myProcId ) {
        stk::CommBuffer& proc_buff = commSparse.send_buffer(proc);
        proc_buff.pack<double>(sendSomeNumber);
      }
    }
  });

  for (int proc=0;proc<numProcs;proc++) {
    if ( proc != myProcId ) {
      stk::CommBuffer& dataReceived = commSparse.recv_buffer(proc);
      double val = -1;
      dataReceived.unpack(val);
      EXPECT_EQ(100-proc, val);
    }
  }
}

TEST(ParallelComm, HowToCommunicateOneValue_RawCommSparse)
{
  MPI_Comm comm = MPI_COMM_WORLD;
  stk::CommSparse commSparse(comm);

  int myProcId = commSparse.parallel_rank();
  int numProcs = commSparse.parallel_size();

  double sendSomeNumber = 100-myProcId;

  for(int phase = 0; phase < 2; ++phase) {
    for (int proc=0;proc<numProcs;proc++) {
      if ( proc != myProcId ) {
        stk::CommBuffer& proc_buff = commSparse.send_buffer(proc);
        proc_buff.pack<double>(sendSomeNumber);
      }
    }
    if(phase == 0) {
      commSparse.allocate_buffers();
    }
    else {
      commSparse.communicate();
    }
  }

  for (int proc=0;proc<numProcs;proc++) {
    if ( proc != myProcId ) {
      stk::CommBuffer& dataReceived = commSparse.recv_buffer(proc);
      double val = -1;
      dataReceived.unpack(val);
      EXPECT_EQ(100-proc, val);
    }
  }
}
```

Listing 2.4 shows how to receive an unknown amount of data from a processor.

**Listing 2.4 Example showing how to communicate an arbitrary amount of data with other processors code/stk/stk_doc_tests/stk_util/CommSparseHowTo.cpp**

```
TEST(ParallelComm, HowToCommunicateAnArbitraryNumberOfValues)
{
  MPI_Comm comm = MPI_COMM_WORLD;
  stk::CommSparse commSparse(comm);

  int myProcId = commSparse.parallel_rank();
  int numProcs = commSparse.parallel_size();

  double sendSomeNumber = 100-myProcId;

  stk::pack_and_communicate(commSparse, [&commSparse, &sendSomeNumber, &myProcId,
           &numProcs]() {
    for (int proc=0;proc<numProcs;proc++) {
      if ( proc != myProcId ) {
        stk::CommBuffer& proc_buff = commSparse.send_buffer(proc);
        for (int i=0;i<myProcId;i++) {
          proc_buff.pack<double>(sendSomeNumber+i);
        }
      }
    }
  });

  for (int sourceProc=0; sourceProc < numProcs; sourceProc++) {
    if ( sourceProc != myProcId ) {
      stk::CommBuffer& dataReceived = commSparse.recv_buffer(sourceProc);
      int numItemsReceived = 0;
      while ( dataReceived.remaining() ) {
        double val = -1;
        dataReceived.unpack(val);
        EXPECT_EQ(100-sourceProc+numItemsReceived, val);
        numItemsReceived++;
      }
      int goldNumItemsReceived = sourceProc;
      EXPECT_EQ(goldNumItemsReceived, numItemsReceived);
    }
  }
}
```

## 2.2. Using the STK Scheduler

The STK Scheduler provides a capability for scheduling an operation, for example output, that will happen at various periods throughout an analysis. The application can create a scheduler and then set the schedule based on time intervals, explicit times, step intervals, and explicit steps. Multiple scheduling intervals can be specified with different scheduling in each interval. The application can then query the scheduler throughout the analysis and determine whether the scheduled activity should be performed at the current analysis time and step.

This section describes two methods of using the STK Scheduler tool: time-based and step-based scheduling. Examples of time-based and step-based scheduling are provided below to show the behavior of the two methods and the combinations thereof. The figures at the end of the section show differences between time-based and step-based scheduling. One main difference is that with time-based scheduling, the `is_it_time()` function will return ‘true” the first time it is called per time period, while the step-based scheduling will return “true” only if the step number is equal to a step period.

In addition to time-based and step-based scheduling, the STK Scheduler state can also be modified via operating system signals and explicit application control; examine the source code to see these additional capabilities.

**Listing 2.5 Using the scheduler**
**code/stk/stk_doc_tests/stk_util/usingScheduler.cpp**

```
#include "gtest/gtest.h"
#include "stk_util/environment/Scheduler.hpp"  // for Scheduler, Time, Step

namespace
{
TEST(StkUtilTestForDocumentation, TimeBasedScheduling)
{
  stk::util::Scheduler scheduler;

  const stk::util::Time startTime = 0.0;
  const stk::util::Time timeInterval = 1.0;
  scheduler.add_interval(startTime, timeInterval);

  stk::util::Step timeStep = 0;
  EXPECT_TRUE(scheduler.is_it_time(0.0, timeStep++));
  EXPECT_FALSE(scheduler.is_it_time(0.5, timeStep++));
  EXPECT_TRUE(scheduler.is_it_time(1.0, timeStep++));
}

TEST(StkUtilTestForDocumentation, TimeBasedSchedulingWithTerminationTime)
{
  stk::util::Scheduler scheduler;

  const stk::util::Time startTime = 2.0;
  const stk::util::Time timeInterval = 10.0;
  scheduler.add_interval(startTime, timeInterval);

  const stk::util::Time terminationTime = 8.2;
  scheduler.set_termination_time(terminationTime);

  stk::util::Step timeStep = 0;
  EXPECT_FALSE(scheduler.is_it_time(startTime - 1.0, timeStep++));
  const stk::util::Time firstTimeAfterStartTime = terminationTime-0.1;
  EXPECT_TRUE(scheduler.is_it_time(firstTimeAfterStartTime, timeStep++));
  const stk::util::Time firstAfterTermination = terminationTime+0.1;
  EXPECT_TRUE(scheduler.is_it_time(firstAfterTermination, timeStep++));
  EXPECT_FALSE(scheduler.is_it_time(terminationTime+0.2, timeStep++));
}

TEST(StkUtilTestForDocumentation, StepBasedScheduler)
{
  stk::util::Scheduler scheduler;

  const stk::util::Step startStep = 0;
  const stk::util::Step stepInterval = 4;
  scheduler.add_interval(startStep, stepInterval);

  const stk::util::Time dt = 0.1;
  for (stk::util::Step timeStep=0;timeStep<100;timeStep+=3)
  {
    stk::util::Time time = timeStep*dt;
    bool check = scheduler.is_it_time(time, timeStep);
    if ( timeStep % stepInterval == 0 )
    {
      EXPECT_TRUE(check);
    }
    else
    {
      EXPECT_FALSE(check);
```

```
    }
  }
}

TEST(StkUtilTestForDocumentation, TimeBasedSchedulerWithTwoTimeIntervals)
{
  stk::util::Scheduler scheduler;
  const stk::util::Time startTime1 = 0.0;
  const stk::util::Time delta1 = 0.1;
  scheduler.add_interval(startTime1, delta1);
  const stk::util::Time startTime2 = 0.9;
  const stk::util::Time delta2 = 0.3;
  scheduler.add_interval(startTime2, delta2);

  stk::util::Step timeStep = 0;
  EXPECT_TRUE(scheduler.is_it_time(0.0, timeStep++));
  EXPECT_FALSE(scheduler.is_it_time(0.07, timeStep++));
  EXPECT_TRUE(scheduler.is_it_time(0.14, timeStep++));
  EXPECT_TRUE(scheduler.is_it_time(0.62, timeStep++));
  EXPECT_TRUE(scheduler.is_it_time(0.6999999, timeStep++));
  EXPECT_FALSE(scheduler.is_it_time(0.77, timeStep++));
  EXPECT_TRUE(scheduler.is_it_time(0.9, timeStep++));
  EXPECT_FALSE(scheduler.is_it_time(0.97, timeStep++));
  EXPECT_FALSE(scheduler.is_it_time(1.04, timeStep++));
  EXPECT_FALSE(scheduler.is_it_time(1.11, timeStep++));
  EXPECT_TRUE(scheduler.is_it_time(1.27, timeStep++));
}
}
```

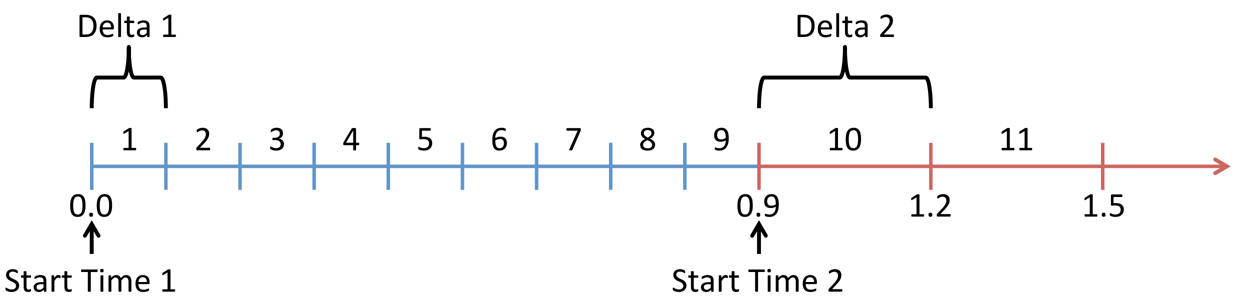



**Figure 2-1. Example time-based scheduler: Using two intervals of different sizes. The first interval spans the time from 0.0 to 0.9 with a time-delta of 0.1; the second interval continues from time 0.9 to the end of the analysis with a time-delta of 0.3.**

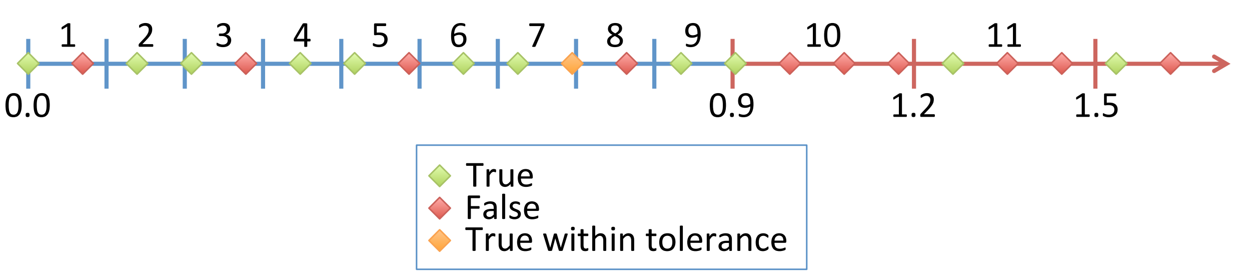



**Figure 2-2. Example time-based scheduler: The first call to `is_it_time()` per interval (within a tolerance) will return true. The diamond shapes show the sequence of calls and the color of the diamond signifies whether the function returns true (green or yellow) or false (red). The time-delta and interval settings are the same as in the previous figure.**

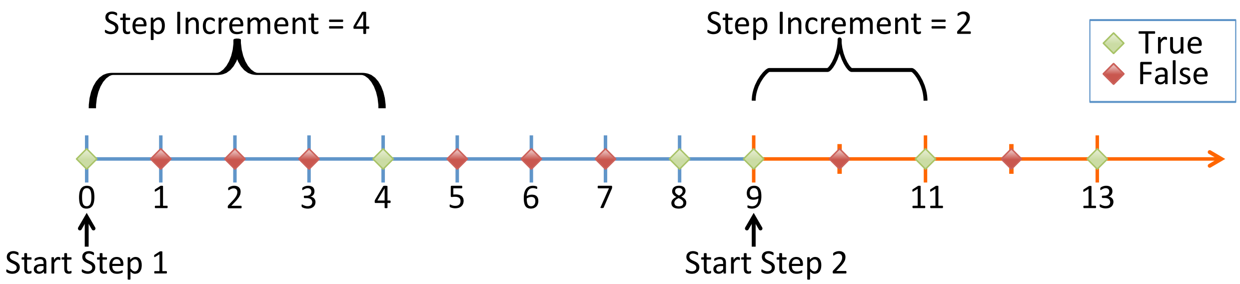



**Figure 2-3. Example step-based scheduler: The call to `is_it_time()` will return true on the interval boundary aligned with the step increment. The diamond shapes show the sequence of calls and the color of the diamond signifies whether the function returns true (green) or false (red). This scheduler has two intervals; the first spans steps 0 to 9 with a step-increment of 4 followed by an interval with a step-increment of 2.**

## 2.3. Parameters – type-safe named storage of any variable type

The `Parameter` class provides a type-save mechanism for storing any variable. A variable or vector of variables can be stored in a `ParameterList` and later retrieved by name. The parameters can also be read from and written to mesh and results files as demonstrated in Sections 5.1.30 and 5.1.31.

The supported variable types that can currently be stored in a `Parameter` object are 32-bit integers, 64-bit integers, doubles, floats, and std::strings and vectors of those types. If an additional type is required, it can be added fairly easily and non-supported types can be stored with reduced functionality.

The first example sets up some variables of various types for use in the following parameter examples.

**Listing 2.6 Parameters: Data for use in the following examples**
**code/stk/stk_doc_tests/stk_util/parameters.cpp**

```
//+ INITIALIZATION
std::vector<std::string> expected_name;
std::vector<stk::util::ParameterType::Type> expected_type;

//+ Scalar values of type double, float, int, int64_t, and string
double pi = 3.14159;
float e = 2.71828;
int answer = 42;
int64_t big_answer = 42000000000001;
std::string team_name = "STK Transition Team";

expected_name.push_back("PI");
expected_type.push_back(stk::util::ParameterType::DOUBLE);
expected_name.push_back("E");
expected_type.push_back(stk::util::ParameterType::FLOAT);
expected_name.push_back("Answer");
expected_type.push_back(stk::util::ParameterType::INTEGER);
expected_name.push_back("Answer_64");
expected_type.push_back(stk::util::ParameterType::INT64);
expected_name.push_back("TeamName");
expected_type.push_back(stk::util::ParameterType::STRING);

//+ vector of doubles
std::vector<double> my_double_vector;
```

```
my_double_vector.push_back(2.78); my_double_vector.push_back(5.30);
my_double_vector.push_back(6.21);
expected_name.push_back("some_doubles");
expected_type.push_back(stk::util::ParameterType::DOUBLEVECTOR);

//+ vector of floats
std::vector<float> my_float_vector;
my_float_vector.push_back(194.0); my_float_vector.push_back(-194.0);
my_float_vector.push_back(47.0);  my_float_vector.push_back(92.0);
expected_name.push_back("some_floats");
expected_type.push_back(stk::util::ParameterType::FLOATVECTOR);

//+ vector of ints
std::vector<int> ages;
ages.push_back(55); ages.push_back(49); ages.push_back(21); ages.push_back(19);
expected_name.push_back("Ages");
expected_type.push_back(stk::util::ParameterType::INTEGERVECTOR);

//+ vector of int64_ts
std::vector<int64_t> ages_64;
ages_64.push_back(55); ages_64.push_back(49); ages_64.push_back(21); ages_64.push_back(19);
expected_name.push_back("Ages_64");
expected_type.push_back(stk::util::ParameterType::INT64VECTOR);

//+ vector of strings
std::vector<std::string> names;
names.push_back("greg"); names.push_back("chloe"); names.push_back("tuffy");
names.push_back("liberty"); names.push_back("I have spaces");
expected_name.push_back("Names");
expected_type.push_back(stk::util::ParameterType::STRINGVECTOR);

```

This example illustrates how to create a `ParameterList` and add variables to it. Note that a single `ParameterList` can store multiple variables of multiple types.

**Listing 2.7 Parameters: Defining**
**code/stk/stk_doc_tests/stk_util/parameters.cpp**

```
//+ Define parameters...
stk::util::ParameterList params;
params.set_param("PI",           pi);
params.set_param("E",            e);
params.set_param("Answer",       answer);
params.set_param("Answer_64",    big_answer);
params.set_param("TeamName",     team_name);
params.set_param("some_doubles", my_double_vector);
params.set_param("some_floats",  my_float_vector);
params.set_param("Ages",         ages);
params.set_param("Ages_64",      ages_64);
params.set_param("Names",        names);

```

Once the parameters have been added to a `ParameterList`, they can be printed or accessed by various means as shown in the following example.

**Listing 2.8 Parameters: Accessing values**
**code/stk/stk_doc_tests/stk_util/parameters.cpp**

```
//+ Write parameters to stdout...
params.write_parameter_list(std::cout);

//+ Access parameters by name...
size_t num_param = expected_name.size();
```

```
for (size_t i=0; i < num_param; i++) {
  stk::util::Parameter &param = params.get_param(expected_name[i]);
  EXPECT_EQ(param.type, expected_type[i]);
}

//+ Extract some parameter values if know type:
std::vector<int> pages = params.get_value<std::vector<int> >("Ages");
for (size_t i=0; i < pages.size(); i++) {
  EXPECT_EQ(pages[i], ages[i]);
}

double my_pi = params.get_value<double>("PI");
EXPECT_EQ(my_pi, pi);

//+ Change value of an existing parameter
params.set_value("Answer", 21);

int new_answer = params.get_value<int>("Answer");
EXPECT_EQ(new_answer, 21);

{
  //+ Access a variable of unknown type...
  stk::util::Parameter &param = params.get_param("Answer");
  double value_as_double = 0.0;
  switch (param.type) {
  case stk::util::ParameterType::DOUBLE:
    value_as_double = param.get_value<double>();
    break;
  case stk::util::ParameterType::FLOAT:
    value_as_double = static_cast<double>(param.get_value<float>());
    break;
  case stk::util::ParameterType::INTEGER:
    value_as_double = static_cast<double>(param.get_value<int>());
    break;
  case stk::util::ParameterType::INT64:
    value_as_double = static_cast<double>(param.get_value<int64_t>());
    break;
  default:
    std::cerr << "ERROR: I can not convert 'Answers' value to a double\n";
    break;
  }
  EXPECT_EQ(static_cast<double>(new_answer), value_as_double);
}
```

This example shows how the `Parameter` class deals with errors such as accessing nonexistent parameters or specifying the incorrect type for a parameter.

**Listing 2.9 Parameters: Dealing with errors**
**code/stk/stk_doc_tests/stk_util/parameters.cpp**

```
//+ If the requested parameter does not exist,
//+ an error message is printed to stderr and an invalid
//+ parameter object is returned
stk::util::Parameter no_exist = params.get_param("DoesNotExist");
EXPECT_EQ(stk::util::ParameterType::INVALID, no_exist.type);

//+ In this method of requesting a parameter, no error
//+ message is printed if the parameter doesn't exist and
//+ instead the returned iterator is equal to the end of the
//+ parameter list.
stk::util::ParameterMapType::iterator it = params.find("DoesNotExist");
EXPECT_TRUE(it == params.end());

//+ If the value of a non-existant parameter is requested,
```

```
//+ an error message is printed and the value 0 is returned.
double invalid_value = params.get_value<double>("DoesNotExist");
EXPECT_EQ(0.0, invalid_value);

//+ If the parameter types do not match, an error message is
//+ printed and the value 0 of the requested type is returned.
int invalid = params.get_value<int>("PI");
EXPECT_EQ(0, invalid);

//+ If the parameter types do not match, an error message is
//+ printed and an empty vector of the requested type is returned.
std::vector<double> pies = params.get_value<std::vector<double> >("PI");
EXPECT_EQ(0u, pies.size());

```

Although it is best to use a `ParameterList` with the supported variable types, it can also be used to store types that it does not officially support. The following example shows this capability by storing a value of `std::complex` type. Note that although an unsupported type can be stored and retrieved from a `ParameterList`, it cannot be read from or written to a mesh or results file or printed using the `Parameter` system.

**Listing 2.10 Parameters: Storing unsupported types**
**code/stk/stk_doc_tests/stk_util/parameters.cpp**

```
//+ Adding a parameter of "unsupported" type...
stk::util::ParameterList more_params;
std::complex<double> phase(3.14,2.718);
more_params.set_param("phase", phase);

//+ The print system doesn't know about this type, so will print
//+ a warning message about unrecognized type.
more_params.write_parameter_list(std::cout);

//+ However, you can still retrieve the value of the parameter
//+ if you know what type it is.
std::complex<double> my_phase = more_params.get_value<std::complex<double> >("phase");
EXPECT_EQ(my_phase, phase);

//+ The Parameter class won't help you on determining the type,
//+ You must know what it is.
EXPECT_EQ(more_params.get_param("phase").type, stk::util::ParameterType::INVALID);

//+ If the wrong type is specified, an exception will be thrown...
EXPECT_ANY_THROW(more_params.get_value<std::complex<int> >("phase"));

```

## 2.4. Filename substitution

The filename_substitution function in STK Util provides a basic substitution capability. If the string (typically a filename) passed as an argument to this function contains “special characters”, the special characters will be replaced with runtime-calculated values. The currently supported substitutions are:

- `%B` For applications which use the command-line-argument parsing facilities provided in `stk_util/environment/ProgramOptions.hpp`, and which use a command-line argument called “input-deck”, then `%B` will be replaced by the basename of the file named as that “input-deck” argument. If there is no “input-deck” argument, then the basename “stdin” will be used. The basename of the file is the portion of the string between the last “/” and the last “.”. For example, given the string `/path/to/the/file/`**input**`.i`, the basename would be `input`.

- `%P` will be replaced by the number of processors being used in the current execution.

The example below shows a very simple example of this capability. It is run on 1 processor with no input file, so the substituted filename should be “stdin-1.e”.

**Listing 2.11 Filename substitution capability**
**code/stk/stk_doc_tests/stk_util/filenameSubstitution.cpp**

```
#include "gtest/gtest.h"
#include "stk_util/environment/EnvData.hpp"          // for EnvData
#include "stk_util/environment/Env.hpp"
#include "stk_util/environment/FileUtils.hpp"        // for filename_substitution
#include "stk_util/environment/ParsedOptions.hpp"    // for ParsedOptions
#include "stk_util/environment/ProgramOptions.hpp"   // for get_parsed_options
#include <string>                                    // for allocator, operator+, string,
            char_tr...

namespace
{
TEST(StkUtilHowTo, useFilenameSubstitutionWithNoCommandLineOptions)
{
  const std::string default_base_filename = "stdin";
  const int numProcs = stk::parallel_machine_size(sierra::Env::parallel_comm());
  const std::string numProcsString = std::to_string(numProcs);
  const std::string expected_filename = default_base_filename + "-" + numProcsString + ".e";

  std::string file_name = "%B-%P.e";
  stk::util::filename_substitution(file_name);
  EXPECT_EQ(expected_filename, file_name);
}

void setFilenameInCommandLineOptions(const std::string &filename)
{
  stk::get_parsed_options().insert("input-deck", filename);
  stk::EnvData::instance().m_inputFile = filename;
}

TEST(StkUtilHowTo, useFilenameSubstitutionWithFileComingFromCommandLineOptions)
{
  const std::string base_filename = "myfile";
  const std::string full_filename = "/path/to/" + base_filename + ".g";
  setFilenameInCommandLineOptions(full_filename);

  const int numProcs = stk::parallel_machine_size(sierra::Env::parallel_comm());
  const std::string numProcsString = std::to_string(numProcs);
  const std::string expected_filename = base_filename + "-" + numProcsString + ".e";

  std::string file_name = "%B-%P.e";
  stk::util::filename_substitution(file_name);

  EXPECT_EQ(expected_filename, file_name);
}
}
```

## 2.5. Using the Diagnostic Timers

The following tests show the basic usage of the Diagnostic Timers.

**Listing 2.12 Diagnostic Timers**
**code/stk/stk_doc_tests/stk_util/TimerHowTo.cpp**

```
#include "gtest/gtest.h"
#include "stk_unit_test_utils/stringAndNumberComparisons.hpp"  // for
            areStringsEqualWithToleran...
#include "stk_util/diag/PrintTimer.hpp"                        // for printTimersTable
#include "stk_util/diag/Timer.hpp"                             // for Timer, createRootTimer
#include "stk_util/diag/TimerMetricTraits.hpp"                 // for METRICS_ALL
#include <unistd.h>                                            // for usleep
```

```
#include <iosfwd>                                       // for ostringstream
#include <string>                                       // for string

namespace
{

#if defined(NDEBUG)
  const double tolerance = 0.10;
#else
  const double tolerance = 0.25;
#endif

void doWork()
{
  ::usleep(1e5);
}

TEST(StkDiagTimerHowTo, useTheRootTimer)
{
  stk::diag::TimerSet enabledTimerSet(0);
  stk::diag::Timer rootTimer = createRootTimer("totalTestRuntime", enabledTimerSet);

  {
    stk::diag::TimeBlock totalTestRuntime(rootTimer);
    doWork();

    std::ostringstream outputStream;
    bool printTimingsOnlySinceLastPrint = false;
    stk::diag::printTimersTable(outputStream, rootTimer, stk::diag::METRICS_ALL,
             printTimingsOnlySinceLastPrint);

    std::string expectedOutput = "                                                     \
                 Timer                   Count       CPU Time              Wall Time       \
---------------------------------------- ----- --------------------- --------------------- \
totalTestRuntime                           1        SKIP  SKIP            00:00:00.100 SKIP
                 \
                                                                                           \
Took 0.0001 seconds to generate the table above.                                           \
    ";
    using stk::unit_test_util::areStringsEqualWithToleranceForNumbers;
    EXPECT_TRUE(areStringsEqualWithToleranceForNumbers(expectedOutput, outputStream.str(),
             tolerance));
  }

  stk::diag::deleteRootTimer(rootTimer);
}

TEST(StkDiagTimerHowTo, useChildTimers)
{
  enum {CHILDMASK1 = 1, CHILDMASK2 = 2};
  stk::diag::TimerSet enabledTimerSet(CHILDMASK1 | CHILDMASK2);
  stk::diag::Timer rootTimer = createRootTimer("totalTestRuntime", enabledTimerSet);
  rootTimer.start();

  stk::diag::Timer childTimer1("childTimer1", CHILDMASK1, rootTimer);
  stk::diag::Timer childTimer2("childTimer2", CHILDMASK2, rootTimer);

  {
    stk::diag::TimeBlock timeStuffInThisScope(childTimer1);
    stk::diag::TimeBlock timeStuffInThisScopeAgain(childTimer2);
    doWork();
  }

  std::ostringstream outputStream;
  bool printTimingsOnlySinceLastPrint = false;
  stk::diag::printTimersTable(outputStream, rootTimer, stk::diag::METRICS_ALL,
             printTimingsOnlySinceLastPrint);

```

```
  {
    stk::diag::TimeBlock timeStuffInThisScope(childTimer1);
    doWork();
  }

  stk::diag::printTimersTable(outputStream, rootTimer, stk::diag::METRICS_ALL,
             printTimingsOnlySinceLastPrint);

  std::string expectedOutput = "                                                         \
                 Timer                   Count       CPU Time              Wall Time       \
---------------------------------------- ----- --------------------- --------------------- \
totalTestRuntime                             1        SKIP   SKIP        00:00:00.100 SKIP  \
  childTimer1                                1        SKIP   SKIP        00:00:00.100 SKIP  \
  childTimer2                                1        SKIP   SKIP        00:00:00.100 SKIP  \
                                                                                           \
Took 0.0001 seconds to generate the table above.                                           \
                 Timer                   Count       CPU Time              Wall Time       \
---------------------------------------- ----- --------------------- --------------------- \
totalTestRuntime                             1        SKIP   SKIP        00:00:00.200 SKIP  \
  childTimer1                                2        SKIP   SKIP        00:00:00.200 SKIP  \
  childTimer2                                1        SKIP   SKIP        00:00:00.100 SKIP  \
                                                                                           \
Took 0.0001 seconds to generate the table above.                                           \
             ";
  using stk::unit_test_util::areStringsEqualWithToleranceForNumbers;
  EXPECT_TRUE(areStringsEqualWithToleranceForNumbers(expectedOutput, outputStream.str(),
             tolerance));

  stk::diag::deleteRootTimer(rootTimer);
}

TEST(StkDiagTimerHowTo, disableChildTimers)
{
  enum {CHILDMASK1 = 1, CHILDMASK2 = 2};
  stk::diag::TimerSet enabledTimerSet(CHILDMASK2);
  stk::diag::Timer rootTimer = createRootTimer("totalTestRuntime", enabledTimerSet);
  rootTimer.start();

  stk::diag::Timer disabledTimer("disabledTimer", CHILDMASK1, rootTimer);
  stk::diag::Timer enabledTimer("enabledTimer", CHILDMASK2, rootTimer);

  {
    stk::diag::TimeBlock timeStuffInThisScope(disabledTimer);
    stk::diag::TimeBlock timeStuffInThisScopeAgain(enabledTimer);
    doWork();
  }

  std::ostringstream outputStream;
  bool printTimingsOnlySinceLastPrint = false;
  stk::diag::printTimersTable(outputStream, rootTimer, stk::diag::METRICS_ALL,
             printTimingsOnlySinceLastPrint);

  {
    stk::diag::TimeBlock timeStuffInThisScope(disabledTimer);
    doWork();
  }

  stk::diag::printTimersTable(outputStream, rootTimer, stk::diag::METRICS_ALL,
             printTimingsOnlySinceLastPrint);

  std::string expectedOutput = "                                                         \
                 Timer                   Count       CPU Time              Wall Time       \
```

```
---------------------------------------- ----- --------------------- --------------------- \
totalTestRuntime                             1         SKIP   SKIP        00:00:00.100 SKIP                 \
  enabledTimer                               1         SKIP   SKIP        00:00:00.100 SKIP                 \
                                                                                          \
Took 0.0001 seconds to generate the table above.                                          \
                 Timer                   Count       CPU Time              Wall Time       \
---------------------------------------- ----- --------------------- --------------------- \
totalTestRuntime                             1         SKIP   SKIP        00:00:00.200 SKIP                 \
  enabledTimer                               1         SKIP   SKIP        00:00:00.100 SKIP                 \
                                                                                          \
Took 0.0001 seconds to generate the table above.                                          \
            ";
  using stk::unit_test_util::areStringsEqualWithToleranceForNumbers;
  EXPECT_TRUE(areStringsEqualWithToleranceForNumbers(expectedOutput, outputStream.str(),
             tolerance));

  stk::diag::deleteRootTimer(rootTimer);
}

}
```

**Listing 2.13 Diagnostic Timers in Parallel**
**code/stk/stk_doc_tests/stk_util/TimerHowToParallel.cpp**

```
#include "gtest/gtest.h"
#include "stk_unit_test_utils/stringAndNumberComparisons.hpp"  // for
            areStringsEqualWithToleran...
#include "stk_util/diag/PrintTimer.hpp"                        // for printTimersTable
#include "stk_util/diag/Timer.hpp"                             // for Timer, createRootTimer
#include "stk_util/diag/TimerMetricTraits.hpp"                 // for METRICS_ALL
#include "stk_util/parallel/Parallel.hpp"                      // for MPI_Comm_rank,
            MPI_Comm_size
#include <unistd.h>                                            // for usleep
#include <iosfwd>                                              // for ostringstream
#include <string>                                              // for string

namespace
{

const double tolerance = 0.10;

void doWork()
{
  ::usleep(1e5);
}

TEST(StkDiagTimerHowTo, useTimersInParallel)
{
  stk::ParallelMachine communicator = MPI_COMM_WORLD;
  int numProcs = stk::parallel_machine_size(communicator);
  if(numProcs == 2)
  {
    enum {CHILDMASK1 = 1};
    stk::diag::TimerSet enabledTimerSet(CHILDMASK1);
    stk::diag::Timer rootTimer = createRootTimer("totalTestRuntime", enabledTimerSet);
    rootTimer.start();

    stk::diag::Timer childTimer1("childTimer1", CHILDMASK1, rootTimer);

    {
      stk::diag::TimeBlockSynchronized timerStartSynchronizedAcrossProcessors(childTimer1,
             communicator);
```

```
      doWork();
    }

    std::ostringstream outputStream;
    bool printTimingsOnlySinceLastPrint = false;
    stk::diag::printTimersTable(outputStream, rootTimer, stk::diag::METRICS_ALL,
            printTimingsOnlySinceLastPrint, communicator);

    int procId = stk::parallel_machine_rank(communicator);
    if(procId == 0)
    {
      std::string expectedOutput = "                                                   \
                                  CPU Time             CPU Time             CPU Time             \
                                 Wall Time            Wall Time            Wall Time             \
     Timer      Count      Sum (% of System)   Min (% of System)   Max (% of System)         \
                             Sum (% of System)   Min (% of System)   Max (% of System)         \
SKIP ----- --------------------- --------------------- SKIP SKIP SKIP --------------------- \
totalTestRuntime  2            SKIP  SKIP          SKIP  SKIP          SKIP  SKIP           \
                               00:00:00.200 SKIP          00:00:00.100 SKIP
            00:00:00.100 SKIP           \
  childTimer1     2            SKIP  SKIP          SKIP  SKIP          SKIP  SKIP           \
                               00:00:00.200 SKIP          00:00:00.100 SKIP
            00:00:00.100 SKIP           \
                                                                                            \
Took SKIP seconds to generate the table above.                                            \
                   ";
std::cerr<<expectedOutput<<" : "<<outputStream.str()<<std::endl;
      using stk::unit_test_util::areStringsEqualWithToleranceForNumbers;
      EXPECT_TRUE(areStringsEqualWithToleranceForNumbers(expectedOutput, outputStream.str(),
            tolerance));
    }

    stk::diag::deleteRootTimer(rootTimer);
  }
}

}
```

The line at the end that prints the time to generate the table is not that useful for small or medium sized runs, but at large numbers of processors, it can take a non-trivial amount of time to gather the timing data from all processors. Knowing this time can help you understand the overall problem runtime.

# 3. STK TOPOLOGY

As stated in the introductory chapter, *Topology* provides an entity's finite element description and this includes a number of attributes such as the number and type of lower-rank entities that can exist in that entity's downward connectivity (e.g., the number of faces that an element topology can have, the ordering of nodes attached to particular faces, etc.).

A primary goal of `stk_topology` is to provide fast traversal of sub-topologies, such as the edges of an element or the nodes of a face, etc. `stk_topology` uses value semantics (e.g., no pointers to singletons) and can be used on GPUs as well as CPUs. `stk_topology` provides compile-time access to topology information, as well as run-time. (See Section 3.1.3, Listing 3.3).

## 3.1. STK Topology API

This section contains several code listings that attempt to aid in the understanding of the stk topology API.

Note the following details of the API:

- `num_nodes()` vs `num_vertices()`: For linear topologies, the number of nodes equals the number of vertices. For higher order topologies, "nodes" include those located at the corners as well as those located at mide-sides and/or mid-edges; but "vertices" are only those nodes located at the corners.
- `is_shell()`: This is a helper to distinguish between "structural" elements (such as shells and beams), and "continuum" elements.
- Permutations (`num_permutations()` vs `num_positive_permutations()`): Different orderings of a topology's nodes may appear in certain contexts. Positive vs negative refers to whether a given node ordering represents a different direction "normal" for that topology. Note also that this isn't a true mathematical permutation since not all possible "permutations" of the nodes are even valid; these permutations are essentially node traversals with the same sequence but different starting points.
- `base()`: For topologies with polynomial order higher than linear, "base()" provides the corresponding linear topology.
- `is_superelement()`, `create_superelement_topology()`: Super-elements are used for reduced-order modeling in certain application formulations.

### 3.1.1. How to set and get topology

This example shows how to attach topology to entities (if entities are created “in line” rather than being created by STK IO). Essentially, topology is attached to entities by declaring the entities to be members of a Part that has the desired topology. The example also shows how to retrieve topology from the mesh. More detailed information about STK Topology is provided in Chapter 3.

**Listing 3.1 Example of setting/getting topology**
**code/stk/stk_doc_tests/stk_mesh/setAndGetTopology.cpp**

```
TEST(stkMeshHowTo, setAndGetTopology)
{
  const unsigned spatialDimension = 3;
  stk::mesh::MeshBuilder builder(MPI_COMM_WORLD);
  builder.set_spatial_dimension(spatialDimension);
  builder.set_entity_rank_names(stk::mesh::entity_rank_names());
  std::shared_ptr<stk::mesh::BulkData> bulkPtr = builder.create();
  stk::mesh::MetaData& metaData = bulkPtr->mesh_meta_data();
  stk::mesh::Part &tetPart = metaData.declare_part_with_topology("tet part",
            stk::topology::TET_4);

  stk::mesh::Part &hexPart = metaData.declare_part("existing part with currently unknown
            topology");
  // . . . then later assigned
  stk::mesh::set_topology(hexPart, stk::topology::HEX_8);

  metaData.commit();
  stk::mesh::BulkData& bulkData = *bulkPtr;

  bulkData.modification_begin();
  stk::mesh::EntityId elem1Id = 1, elem2Id = 2;
  stk::mesh::Entity elem1 = bulkData.declare_element(elem1Id,
            stk::mesh::ConstPartVector{&tetPart});
  stk::mesh::Entity elem2 = bulkData.declare_element(elem2Id,
            stk::mesh::ConstPartVector{&hexPart});
  declare_element_nodes(bulkData, elem1, elem2);
  bulkData.modification_end();

  stk::topology elem1_topology = bulkData.bucket(elem1).topology();
  stk::topology elem2_topology = bulkData.bucket(elem2).topology();

  EXPECT_EQ(stk::topology::TET_4, elem1_topology);
  EXPECT_EQ(stk::topology::HEX_8, elem2_topology);
}
```

### 3.1.2. STK topology ranks

Listing 3.2 demonstrates the link between various STK topologies and their ranks.

**Listing 3.2 Example showing mapping of STK topologies to ranks**
**code/stk/stk_doc_tests/stk_topology/map_stk_topologies_to_ranks.cpp**

```
TEST(stk_topology_how_to, map_topologies_to_ranks )
{
  stk::topology topology = stk::topology::INVALID_TOPOLOGY;
  EXPECT_EQ(stk::topology::INVALID_RANK, topology.rank());

  std::vector<stk::topology> node_rank_topologies;
```

```
node_rank_topologies.push_back(stk::topology::NODE);

std::vector<stk::topology> edge_rank_topologies;
edge_rank_topologies.push_back(stk::topology::LINE_2);
edge_rank_topologies.push_back(stk::topology::LINE_3);

std::vector<stk::topology> face_rank_topologies;
face_rank_topologies.push_back(stk::topology::TRI_3);
face_rank_topologies.push_back(stk::topology::TRIANGLE_3);
face_rank_topologies.push_back(stk::topology::TRI_4);
face_rank_topologies.push_back(stk::topology::TRIANGLE_4);
face_rank_topologies.push_back(stk::topology::TRI_6);
face_rank_topologies.push_back(stk::topology::TRIANGLE_6);
face_rank_topologies.push_back(stk::topology::QUAD_4);
face_rank_topologies.push_back(stk::topology::QUADRILATERAL_4);
face_rank_topologies.push_back(stk::topology::QUAD_6);
face_rank_topologies.push_back(stk::topology::QUADRILATERAL_6);
face_rank_topologies.push_back(stk::topology::QUAD_8);
face_rank_topologies.push_back(stk::topology::QUADRILATERAL_8);
face_rank_topologies.push_back(stk::topology::QUAD_9);
face_rank_topologies.push_back(stk::topology::QUADRILATERAL_9);

std::vector<stk::topology> element_rank_topologies;
element_rank_topologies.push_back(stk::topology::PARTICLE);
element_rank_topologies.push_back(stk::topology::LINE_2_1D);
element_rank_topologies.push_back(stk::topology::LINE_3_1D);
element_rank_topologies.push_back(stk::topology::BEAM_2);
element_rank_topologies.push_back(stk::topology::BEAM_3);
element_rank_topologies.push_back(stk::topology::SHELL_LINE_2);
element_rank_topologies.push_back(stk::topology::SHELL_LINE_3);
element_rank_topologies.push_back(stk::topology::SPRING_2);
element_rank_topologies.push_back(stk::topology::SPRING_3);

element_rank_topologies.push_back(stk::topology::TRI_3_2D);
element_rank_topologies.push_back(stk::topology::TRIANGLE_3_2D);
element_rank_topologies.push_back(stk::topology::TRI_4_2D);
element_rank_topologies.push_back(stk::topology::TRIANGLE_4_2D);
element_rank_topologies.push_back(stk::topology::TRI_6_2D);
element_rank_topologies.push_back(stk::topology::TRIANGLE_6_2D);
element_rank_topologies.push_back(stk::topology::QUAD_4_2D);
element_rank_topologies.push_back(stk::topology::QUADRILATERAL_4_2D);
element_rank_topologies.push_back(stk::topology::QUAD_8_2D);
element_rank_topologies.push_back(stk::topology::QUADRILATERAL_8_2D);
element_rank_topologies.push_back(stk::topology::QUAD_9_2D);
element_rank_topologies.push_back(stk::topology::QUADRILATERAL_9_2D);

element_rank_topologies.push_back(stk::topology::SHELL_TRI_3);
element_rank_topologies.push_back(stk::topology::SHELL_TRIANGLE_3);
element_rank_topologies.push_back(stk::topology::SHELL_TRI_4);
element_rank_topologies.push_back(stk::topology::SHELL_TRIANGLE_4);
element_rank_topologies.push_back(stk::topology::SHELL_TRI_6);
element_rank_topologies.push_back(stk::topology::SHELL_TRIANGLE_6);

element_rank_topologies.push_back(stk::topology::SHELL_QUAD_4);
element_rank_topologies.push_back(stk::topology::SHELL_QUADRILATERAL_4);
element_rank_topologies.push_back(stk::topology::SHELL_QUAD_8);
element_rank_topologies.push_back(stk::topology::SHELL_QUADRILATERAL_8);
element_rank_topologies.push_back(stk::topology::SHELL_QUAD_9);
element_rank_topologies.push_back(stk::topology::SHELL_QUADRILATERAL_9);

element_rank_topologies.push_back(stk::topology::TET_4);
element_rank_topologies.push_back(stk::topology::TETRAHEDRON_4);
element_rank_topologies.push_back(stk::topology::TET_8);
element_rank_topologies.push_back(stk::topology::TETRAHEDRON_8);
element_rank_topologies.push_back(stk::topology::TET_10);
element_rank_topologies.push_back(stk::topology::TETRAHEDRON_10);
element_rank_topologies.push_back(stk::topology::TET_11);
element_rank_topologies.push_back(stk::topology::TETRAHEDRON_11);
```

```

  element_rank_topologies.push_back(stk::topology::PYRAMID_5);
  element_rank_topologies.push_back(stk::topology::PYRAMID_13);
  element_rank_topologies.push_back(stk::topology::PYRAMID_14);
  element_rank_topologies.push_back(stk::topology::WEDGE_6);
  element_rank_topologies.push_back(stk::topology::WEDGE_12);
  element_rank_topologies.push_back(stk::topology::WEDGE_15);
  element_rank_topologies.push_back(stk::topology::WEDGE_18);
  element_rank_topologies.push_back(stk::topology::QUADRILATERAL_9_2D);
  element_rank_topologies.push_back(stk::topology::QUADRILATERAL_9_2D);

  element_rank_topologies.push_back(stk::topology::HEX_8);
  element_rank_topologies.push_back(stk::topology::HEXAHEDRON_8);
  element_rank_topologies.push_back(stk::topology::HEX_20);
  element_rank_topologies.push_back(stk::topology::HEXAHEDRON_20);
  element_rank_topologies.push_back(stk::topology::HEX_27);
  element_rank_topologies.push_back(stk::topology::HEXAHEDRON_27);

  unsigned num_nodes_in_super_element = 10;
  element_rank_topologies.
      push_back(stk::create_superelement_topology(num_nodes_in_super_element));


```

### 3.1.3. Compile-time STK topology information

Listing 3.3 demonstrates how to access compile-time topology information. In this example, `compiletime_num_nodes` is a variable that is assigned a constant, compile-time value. `compiletime_hex8` is a type of **`struct`**, and `num_nodes` is a static const member whose value is defined at compile-time. It thus can be used to allocate space on the stack instead of on the heap. Other compile-time topology attributes are defined by the members of the `topology::topology_type` struct in the file `stk_topology/topology_type.tcc`.

**Listing 3.3 Example using compile-time STK topology information**
**code/stk/stk_doc_tests/stk_topology/runtime_vs_compiletime_topology.cpp**

```cpp
TEST(stk_topology_how_to, runtime_vs_compiletime_topology )
{
  stk::topology runtime_hex8 = stk::topology::HEX_8;

  typedef stk::topology::topology_type<stk::topology::HEX_8> compiletime_hex8;

  const unsigned compiletime_num_nodes = compiletime_hex8::num_nodes;

  EXPECT_EQ( runtime_hex8.num_nodes(), compiletime_num_nodes );

  //declare a static array with length given by compile-time num-nodes
  double compile_time_sized_array[compiletime_num_nodes];
  EXPECT_EQ(sizeof(compile_time_sized_array), sizeof(double)*compiletime_num_nodes);
}
```

### 3.1.4. STK topology for the Particle

Listing 3.4 demonstrates the API for a Particle element.

**Listing 3.4 Example showing STK topology for a zero-dimensional element**
**code/stk/stk_doc_tests/stk_topology/element_topologies.cpp**

```
TEST(stk_topology_understanding, zero_dim_element)
{
  stk::topology sphere = stk::topology::PARTICLE;

  EXPECT_TRUE(sphere.is_valid());
  EXPECT_FALSE(sphere.has_homogeneous_faces());
  EXPECT_FALSE(sphere.is_shell());

  EXPECT_TRUE(sphere.rank() != stk::topology::NODE_RANK);
  EXPECT_TRUE(sphere.rank() != stk::topology::EDGE_RANK);
  EXPECT_TRUE(sphere.rank() != stk::topology::FACE_RANK);
  EXPECT_TRUE(sphere.rank() != stk::topology::CONSTRAINT_RANK);
  EXPECT_TRUE(sphere.rank() == stk::topology::ELEMENT_RANK);

  EXPECT_EQ(sphere.side_rank(), stk::topology::INVALID_RANK);

  EXPECT_EQ(sphere.dimension(),1u);
  EXPECT_EQ(sphere.num_nodes(),1u);
  EXPECT_EQ(sphere.num_vertices(),1u);
  EXPECT_EQ(sphere.num_edges(),0u);
  EXPECT_EQ(sphere.num_faces(),0u);
  EXPECT_EQ(sphere.num_sides(),0u);
  EXPECT_EQ(sphere.num_permutations(),1u);
  EXPECT_EQ(sphere.num_positive_permutations(),1u);

  EXPECT_FALSE(sphere.defined_on_spatial_dimension(0));

  EXPECT_TRUE(sphere.defined_on_spatial_dimension(1));
  EXPECT_TRUE(sphere.defined_on_spatial_dimension(2));
  EXPECT_TRUE(sphere.defined_on_spatial_dimension(3));

  EXPECT_EQ(sphere.base(),stk::topology::PARTICLE);
}
```

### 3.1.5. STK topology for the high order Beam

Listing 3.5 demonstrates the API for a higher order Beam element.

**Listing 3.5 Example of STK topology for a one-dimensional element**
**code/stk/stk_doc_tests/stk_topology/element_topologies.cpp**

```
TEST(stk_topology_understanding, one_dim_higher_order_element)
{
  stk::topology secondOrderBeam = stk::topology::BEAM_3;

  EXPECT_TRUE(secondOrderBeam.is_valid());
  EXPECT_FALSE(secondOrderBeam.has_homogeneous_faces());
  EXPECT_FALSE(secondOrderBeam.is_shell());

  EXPECT_TRUE(secondOrderBeam.rank() != stk::topology::NODE_RANK);
  EXPECT_TRUE(secondOrderBeam.rank() != stk::topology::EDGE_RANK);
  EXPECT_TRUE(secondOrderBeam.rank() != stk::topology::FACE_RANK);
  EXPECT_TRUE(secondOrderBeam.rank() != stk::topology::CONSTRAINT_RANK);
  EXPECT_TRUE(secondOrderBeam.rank() == stk::topology::ELEMENT_RANK);

  EXPECT_TRUE(secondOrderBeam.side_rank() == stk::topology::EDGE_RANK);

  EXPECT_EQ(2u, secondOrderBeam.dimension());
  EXPECT_EQ(3u, secondOrderBeam.num_nodes());
  EXPECT_EQ(2u, secondOrderBeam.num_vertices());
```

```
  EXPECT_EQ(1u, secondOrderBeam.num_edges());

  EXPECT_EQ(0u, secondOrderBeam.num_faces());
  EXPECT_EQ(1u, secondOrderBeam.num_positive_permutations());
  EXPECT_EQ(2u, secondOrderBeam.num_permutations());

  EXPECT_FALSE(secondOrderBeam.defined_on_spatial_dimension(0));
  EXPECT_FALSE(secondOrderBeam.defined_on_spatial_dimension(1));

  EXPECT_TRUE(secondOrderBeam.defined_on_spatial_dimension(2));
  EXPECT_TRUE(secondOrderBeam.defined_on_spatial_dimension(3));

  EXPECT_TRUE(secondOrderBeam.base() == stk::topology::BEAM_2);

  unsigned beamNodes[3] = { 10, 20, 14 }; // 10 *-------*-------* 20
  //           14
  {
    unsigned expectedNodeOffsets[3] = { 0, 1, 2 };
    //unit-test checking utility:
    checkNodeOrderingAndOffsetsForEdges(secondOrderBeam, beamNodes, expectedNodeOffsets);
  }

  {
    unsigned expectedNodeOffsets[6] = {
      0, 1, 2,
      1, 0, 2
    };

    //unit-test checking utility:
    checkNodeOrderingAndOffsetsForPermutations(secondOrderBeam, beamNodes,
            expectedNodeOffsets);
  }
}
```

### 3.1.6. STK topology for the high order triangular Shell

Listing 3.6 demonstrates the API for a higher order triangular shell element.

**Listing 3.6 Example of STK topology for a two-dimensional element**
**code/stk/stk_doc_tests/stk_topology/element_topologies.cpp**

```
TEST(stk_topology_understanding, two_dim_higher_order_element)
{
  stk::topology secondOrderTriShell = stk::topology::SHELL_TRIANGLE_6;
  EXPECT_TRUE(secondOrderTriShell == stk::topology::SHELL_TRI_6);

  EXPECT_TRUE(secondOrderTriShell.is_valid());
  EXPECT_TRUE(secondOrderTriShell.has_homogeneous_faces());
  EXPECT_TRUE(secondOrderTriShell.is_shell());

  EXPECT_TRUE(secondOrderTriShell.rank() != stk::topology::NODE_RANK);
  EXPECT_TRUE(secondOrderTriShell.rank() != stk::topology::EDGE_RANK);
  EXPECT_TRUE(secondOrderTriShell.rank() != stk::topology::FACE_RANK);
  EXPECT_TRUE(secondOrderTriShell.rank() != stk::topology::CONSTRAINT_RANK);
  EXPECT_TRUE(secondOrderTriShell.rank() == stk::topology::ELEMENT_RANK);

  EXPECT_TRUE(secondOrderTriShell.side_rank() == stk::topology::FACE_RANK);

  EXPECT_EQ(3u, secondOrderTriShell.dimension());
  EXPECT_EQ(6u, secondOrderTriShell.num_nodes());
  EXPECT_EQ(3u, secondOrderTriShell.num_vertices());
  EXPECT_EQ(3u, secondOrderTriShell.num_edges());
  EXPECT_EQ(2u, secondOrderTriShell.num_faces());
```

```
  // permutations are the number of ways the number of vertices can be permuted
  EXPECT_EQ(6u, secondOrderTriShell.num_permutations());
  // positive permutations are ones that the normal is maintained
  EXPECT_EQ(3u, secondOrderTriShell.num_positive_permutations());

  EXPECT_FALSE(secondOrderTriShell.defined_on_spatial_dimension(0));
  EXPECT_FALSE(secondOrderTriShell.defined_on_spatial_dimension(1));
  EXPECT_FALSE(secondOrderTriShell.defined_on_spatial_dimension(2));

  EXPECT_TRUE(secondOrderTriShell.defined_on_spatial_dimension(3));

  EXPECT_TRUE(secondOrderTriShell.base() == stk::topology::SHELL_TRI_3);
  EXPECT_TRUE(secondOrderTriShell.base() == stk::topology::SHELL_TRIANGLE_3);

  unsigned shellNodes[6] = { 10, 11, 12, 100, 101, 102 }; // first 3 are vertex nodes (picture?)

  {
    unsigned goldValuesEdgeOffsets[9] = {
      0, 1, 3,
      1, 2, 4,
      2, 0, 5
    };

    //unit-test checking utility:
    checkNodeOrderingAndOffsetsForEdges(secondOrderTriShell, shellNodes, goldValuesEdgeOffsets);
  }

  {
    unsigned goldValuesFaceNodeOffsets[12] = {
      0, 1, 2, 3, 4, 5,
      0, 2, 1, 5, 4, 3
    };

    //unit-test checking utility:
    checkNodeOrderingAndOffsetsForFaces(secondOrderTriShell, shellNodes, goldValuesFaceNodeOffsets);
  }

  {
    unsigned goldValueOffsetsPerm[36] = {
      0, 1, 2, 3, 4, 5,
      2, 0, 1, 5, 3, 4,
      1, 2, 0, 4, 5, 3,
      0, 2, 1, 5, 4, 3,
      2, 1, 0, 4, 3, 5,
      1, 0, 2, 3, 5, 4
    };

    //unit-test checking utility:
    checkNodeOrderingAndOffsetsForPermutations(secondOrderTriShell, shellNodes, goldValueOffsetsPerm);
  }
}
```

### 3.1.7. STK topology for the linear Hexahedral

Listing 3.7 demonstrates the API for a linear Hexahedral element.

**Listing 3.7 Example of STK topology for a three-dimensional element**
**code/stk/stk_doc_tests/stk_topology/element_topologies.cpp**

```
TEST(stk_topology_understanding, three_dim_linear_element)
{
  stk::topology hex8 = stk::topology::HEX_8;
  EXPECT_TRUE(hex8 == stk::topology::HEXAHEDRON_8);

  EXPECT_TRUE(hex8.is_valid());
  EXPECT_TRUE(hex8.has_homogeneous_faces());
  EXPECT_FALSE(hex8.is_shell());

  EXPECT_TRUE(hex8.rank() != stk::topology::NODE_RANK);
  EXPECT_TRUE(hex8.rank() != stk::topology::EDGE_RANK);
  EXPECT_TRUE(hex8.rank() != stk::topology::FACE_RANK);
  EXPECT_TRUE(hex8.rank() != stk::topology::CONSTRAINT_RANK);
  EXPECT_TRUE(hex8.rank() == stk::topology::ELEMENT_RANK);

  EXPECT_TRUE(hex8.side_rank() == stk::topology::FACE_RANK);

  EXPECT_EQ(3u, hex8.dimension());
  EXPECT_EQ(8u, hex8.num_nodes());
  EXPECT_EQ(8u, hex8.num_vertices());
  EXPECT_EQ(12u, hex8.num_edges());
  EXPECT_EQ(6u, hex8.num_faces());

  if (stk::topology::topology_type<stk::topology::HEX_8>::num_permutations > 1) {
    // permutations are the number of ways the number of vertices can be permuted
    EXPECT_EQ(24u, hex8.num_permutations());
    // positive permutations are ones that the normal is maintained
    EXPECT_EQ(24u, hex8.num_positive_permutations());
  }

  EXPECT_FALSE(hex8.defined_on_spatial_dimension(0));
  EXPECT_FALSE(hex8.defined_on_spatial_dimension(1));
  EXPECT_FALSE(hex8.defined_on_spatial_dimension(2));

  EXPECT_TRUE(hex8.defined_on_spatial_dimension(3));

  EXPECT_TRUE(hex8.base() == stk::topology::HEX_8);

  unsigned hex8Nodes[8] = { 0, 1, 2, 3, 4, 5, 6, 7 };

  {
    for(unsigned i = 0; i < hex8.num_edges(); i++) {
      EXPECT_EQ(hex8.edge_topology(i), stk::topology::LINE_2);
      ASSERT_EQ(hex8.edge_topology(i).num_nodes(), 2u);
    }

    unsigned goldValuesEdgeOffsets[24] = {
      0, 1,
      1, 2,
      2, 3,
      3, 0,
      4, 5,
      5, 6,
      6, 7,
      7, 4,
      0, 4,
      1, 5,
      2, 6,
      3, 7
    };

    //unit-test checking utility:
    checkNodeOrderingAndOffsetsForEdges(hex8, hex8Nodes, goldValuesEdgeOffsets);
  }

  {
    stk::topology goldFaceTopology = stk::topology::QUAD_4;
    unsigned goldNumNodesPerFace = 4;
```

```
    for (unsigned faceIndex=0;faceIndex<hex8.num_faces();faceIndex++)
    {
      EXPECT_EQ(goldFaceTopology, hex8.face_topology(faceIndex));
      ASSERT_EQ(goldNumNodesPerFace, hex8.face_topology(faceIndex).num_nodes());
    }

    unsigned goldValuesFaceOffsets[24] = {
      0, 1, 5, 4,
      1, 2, 6, 5,
      2, 3, 7, 6,
      0, 4, 7, 3,
      0, 3, 2, 1,
      4, 5, 6, 7
    };

    //unit-test checking utility:
    checkNodeOrderingAndOffsetsForFaces(hex8, hex8Nodes, goldValuesFaceOffsets);
  }

}
```

### 3.1.8. STK topology `equivalent` method

Listing 3.8 demonstrates the API for checking, given the nodes of topology, if two entities are equivalent. *The support for HEX_8, etc., only includes positive node-permutations, since there is no current need for negative permutations.*

**Listing 3.8 Example using of an `equivalent` method**
**code/stk/stk_doc_tests/stk_topology/equivalent.cpp**

```
TEST(stk_topology_understanding, equivalent_elements)
{
  stk::EquivalentPermutation areElementsEquivalent;

  {
    if (stk::topology::topology_type<stk::topology::HEX_8>::num_permutations > 1) {
      unsigned hex1[8] = { 0, 1, 2, 3, 4, 5, 6, 7 };
      unsigned hex2[8] = { 4, 7, 6, 5, 0, 3, 2, 1 };
      unsigned hex3[8] = { 4, 5, 6, 7, 0, 1, 2, 3 };

      stk::topology hex8 = stk::topology::HEX_8;

      areElementsEquivalent = hex8.is_equivalent((unsigned*)hex1, (unsigned*)hex2);
      EXPECT_TRUE(areElementsEquivalent.is_equivalent);
      areElementsEquivalent = hex8.is_equivalent((unsigned*)hex1, (unsigned*)hex3);
      EXPECT_FALSE(areElementsEquivalent.is_equivalent);
    }
  }

  {
    unsigned triangle_1[3] = {0, 1, 2};
    unsigned triangle_2[3] = {0, 2, 1};

    stk::topology triangular_shell = stk::topology::SHELL_TRIANGLE_3;

    areElementsEquivalent = triangular_shell.is_equivalent((unsigned*)triangle_1,
            (unsigned*)triangle_2);

    EXPECT_TRUE(areElementsEquivalent.is_equivalent);

    unsigned permutation_index = areElementsEquivalent.permutation_number;
    unsigned goldValue = 3;
```

```
    EXPECT_EQ(goldValue, permutation_index); // From previous unit test, this is the 4th permutation
  }

}
```

### 3.1.9. STK topology's `is_positive_polarity` method

**Listing 3.9 Example using `is_positive_polarity`**
**code/stk/stk_doc_tests/stk_topology/how_to_use_stk_topology.cpp**

```
TEST(stk_topology_how_to, check_for_positive_polarity)
{
  stk::topology quad4Topology = stk::topology::QUAD_4;

  ASSERT_EQ(8u, quad4Topology.num_permutations());
  ASSERT_EQ(4u, quad4Topology.num_positive_permutations());

  EXPECT_TRUE( quad4Topology.is_positive_polarity(0));
  EXPECT_TRUE( quad4Topology.is_positive_polarity(1));
  EXPECT_TRUE( quad4Topology.is_positive_polarity(2));
  EXPECT_TRUE( quad4Topology.is_positive_polarity(3));
  EXPECT_TRUE(!quad4Topology.is_positive_polarity(4));
  EXPECT_TRUE(!quad4Topology.is_positive_polarity(5));
  EXPECT_TRUE(!quad4Topology.is_positive_polarity(6));
  EXPECT_TRUE(!quad4Topology.is_positive_polarity(7));

  //or, print it and examine the output:
  stk::verbose_print_topology(std::cout, quad4Topology);
}
```

### 3.1.10. STK topology's `lexicographical_smallest_permutation` method

Listing 3.10 demonstrates the API for obtaining the smallest lexicographical permutation index. *The support for HEX_8, etc., only includes positive node-permutations.*

**Listing 3.10 Example using `lexicographical_smallest_permutation`**
**code/stk/stk_doc_tests/stk_topology/how_to_use_stk_topology.cpp**

```
TEST(stk_topology_understanding, lexicographical_smallest_permutation)
{
  {
    unsigned triangle_node_ids[3] = {10, 8, 12};

    stk::topology triangular_shell = stk::topology::SHELL_TRIANGLE_3;

    unsigned gold_triangle_permutations[18]= {
      10, 8, 12,
      12, 10, 8,
      8, 12, 10, // lexicographical smallest permutation by node ids if considering only positive permutations
      10, 12, 8,
      12, 8, 10,
      8, 10, 12  // lexicographical smallest permutation by node ids if considering all permutations
    };

    verifyPermutationsForTriangle(triangle_node_ids, gold_triangle_permutations);

    bool usePositivePermutationsOnly = false;
```

```
    unsigned permutation_index =
            triangular_shell.lexicographical_smallest_permutation(triangle_node_ids,
            usePositivePermutationsOnly);
    unsigned gold_lexicographical_smallest_permutation_index = 5;
    // driven by vertices, NOT mid-edge nodes
    EXPECT_EQ(gold_lexicographical_smallest_permutation_index, permutation_index);

    usePositivePermutationsOnly = true;
    permutation_index =
            triangular_shell.lexicographical_smallest_permutation(triangle_node_ids,
            usePositivePermutationsOnly);
    gold_lexicographical_smallest_permutation_index = 2;
    // driven by vertices, NOT mid-edge nodes
    EXPECT_EQ(gold_lexicographical_smallest_permutation_index, permutation_index);
  }
}
```

### 3.1.11. STK topology's `lexicographical smallest permutation preserve polarity` method

Listing 3.11 demonstrates the API for obtaining the smallest lexicographical permutation index that matches the polarity of the input permutation

**Listing 3.11 Example using `lexicographical_smallest_permutation_preserve_polarity`**
**code/stk/stk_doc_tests/stk_topology/how_to_use_stk_topology.cpp**

```
TEST(stk_topology_understanding, lexicographical_smallest_permutation_preserve_polarity)
{
  {
    stk::topology triangular_shell = stk::topology::SHELL_TRIANGLE_3;
    unsigned shell_node_ids[3] = {10, 8, 12};
    {
      unsigned triangle_node_ids[3] = {12, 10, 8};

      unsigned permutation_index =
            triangular_shell.lexicographical_smallest_permutation_preserve_polarity(
            triangle_node_ids, shell_node_ids);
      unsigned expected_positive_permutation = 2;

      EXPECT_EQ(expected_positive_permutation, permutation_index);
      EXPECT_LT(expected_positive_permutation, triangular_shell.num_positive_permutations());
    }
    {
      unsigned triangle_node_ids[3] = {12, 8, 10};

      unsigned permutation_index =
            triangular_shell.lexicographical_smallest_permutation_preserve_polarity(
            triangle_node_ids, shell_node_ids);
      unsigned expected_negative_permutation = 5;

      EXPECT_EQ(expected_negative_permutation, permutation_index);
      EXPECT_GE(expected_negative_permutation, triangular_shell.num_positive_permutations());
    }
  }
}

TEST(stk_topology_understanding, quad_lexicographical_smallest_permutation_preserve_polarity)
{
  {
    stk::topology quad_shell = stk::topology::SHELL_QUAD_4;
    unsigned shell_node_ids[4] = {1, 2, 3, 4};
    {
```

```
      unsigned quad_node_ids[4] = {1, 2, 3, 4};

      unsigned permutation_index =
            quad_shell.lexicographical_smallest_permutation_preserve_polarity(quad_node_ids,
            shell_node_ids);
      unsigned expected_positive_permutation = 0;

      EXPECT_EQ(expected_positive_permutation, permutation_index);
      EXPECT_LT(expected_positive_permutation, quad_shell.num_positive_permutations());
    }

    {
      unsigned quad_node_ids[4] = {1, 4, 3, 2};

      unsigned permutation_index =
            quad_shell.lexicographical_smallest_permutation_preserve_polarity(quad_node_ids,
            shell_node_ids);
      unsigned expected_negative_permutation = 4;

      EXPECT_EQ(expected_negative_permutation, permutation_index);
      EXPECT_GE(expected_negative_permutation, quad_shell.num_positive_permutations());
    }

    {
      unsigned quad_node_ids[4] = {4, 2, 3, 1};

      unsigned permutation_index =
            quad_shell.lexicographical_smallest_permutation_preserve_polarity(quad_node_ids,
            shell_node_ids);
      unsigned expected_invalid_permutation = 8;

      EXPECT_EQ(expected_invalid_permutation, permutation_index);
      EXPECT_EQ(expected_invalid_permutation, quad_shell.num_permutations());
    }
  }
}
```

### 3.1.12. STK Topology's `sub_topology` methods

Listing 3.12 demonstrates the API for obtaining information about a topology's sub-topologies (sub-topologies define downward-connected entities; e.g., the face-rank sub-topology of HEX_20 is QUAD_8.).

**Listing 3.12 Example using of `sub_topology`**
**code/stk/stk_doc_tests/stk_topology/how_to_use_stk_topology.cpp**

```
TEST(stk_topology_understanding, sub_topology)
{
  stk::topology hex20 = stk::topology::HEX_20;
  unsigned hex20Nodes[20] = {
    0, 1, 2, 3,
    4, 5, 6, 7,
    8, 9, 10, 11,
    12, 13, 14, 15,
    16, 17, 18, 19
  };

  unsigned numFaces = hex20.num_sub_topology(stk::topology::FACE_RANK);
  EXPECT_EQ(6u, numFaces);

  unsigned faceIndex=2;
  stk::topology top = hex20.sub_topology(stk::topology::FACE_RANK, faceIndex);
```

```
  EXPECT_EQ(stk::topology::QUADRILATERAL_8, top);

  unsigned nodeIdsFace[8];
  hex20.sub_topology_nodes(hex20Nodes, stk::topology::FACE_RANK, faceIndex, nodeIdsFace);

  unsigned goldIdsFace[8] = { 2, 3, 7, 6, 10, 15, 18, 14 };
  for (unsigned i=0;i<hex20.face_topology(faceIndex).num_nodes();i++)
  {
    EXPECT_EQ(goldIdsFace[i], nodeIdsFace[i]);
  }
}
```

### 3.1.13. STK Topology's `sides` methods

Listing 3.13 demonstrates the API for understanding sides in STK topologies. Note that for some topologies, *sides* differs in meaning from the Exodus [1] standard. For example, the number of sides on a shell-4 in Exodus is 6 (two faces, 4 edges). While `num_sides()` for the SHELL_QUAD_4 in stk_topology also returns 6, all *sides* are entities of face rank (6 faces).

**Listing 3.13 Example for understanding sides in STK topology**
**code/stk/stk_doc_tests/stk_topology/how_to_use_stk_topology.cpp**

```
TEST(stk_topology_understanding, sides)
{
  stk::topology hex20 = stk::topology::HEX_20;
  EXPECT_EQ(6u, hex20.num_sides());

  stk::topology quad8 = stk::topology::SHELL_QUADRILATERAL_8;
  EXPECT_EQ(6u, quad8.num_sides());

  stk::topology wedge = stk::topology::WEDGE_15;
  EXPECT_EQ(5u, wedge.num_sides());
  EXPECT_EQ(stk::topology::QUADRILATERAL_8, wedge.side_topology(0));
  EXPECT_EQ(stk::topology::QUADRILATERAL_8, wedge.side_topology(1));
  EXPECT_EQ(stk::topology::QUADRILATERAL_8, wedge.side_topology(2));
  EXPECT_EQ(stk::topology::TRIANGLE_6, wedge.side_topology(3));
  EXPECT_EQ(stk::topology::TRIANGLE_6, wedge.side_topology(4));

}
```

#### 3.1.13.1. Shell Sides

Some shell topologies carry additional `shell_sides` in their properties. The `shell_sides` are 3D face rank entities that are unique to 3D shells topologies (i.e. SHELL_TRI_3 or SHELL_QUAD_4). While the rank of `shell_sides` matches the rank returned by `side_rank()` on a topology, `shell_sides` are not considered in `num_faces()` or `num_edges()`.

**Listing 3.14 Example for understanding shell sides in STK topology**
**code/stk/stk_doc_tests/stk_topology/shell_sides.cpp**

```
TEST(stk_topology, shell_side_num_sides) {
  stk::topology shell = stk::topology::SHELL_QUAD_4;

  EXPECT_TRUE(shell.is_valid());
```

```
  EXPECT_TRUE(shell.is_shell());

  EXPECT_EQ(shell.rank(),stk::topology::ELEMENT_RANK);
  EXPECT_EQ(shell.side_rank(),stk::topology::FACE_RANK);

  EXPECT_EQ(shell.num_vertices(),4u);
  EXPECT_EQ(shell.num_edges(),4u);

  EXPECT_EQ(shell.num_faces(),2u);
  EXPECT_EQ(shell.num_sides(),6u);
  EXPECT_EQ(shell.num_sub_topology(shell.side_rank()), 2u);
  EXPECT_NE(shell.num_sub_topology(shell.side_rank()), shell.num_sides());
}
```

### 3.1.14. STK Topology's `side_topology` methods

Listing 3.15 demonstrates the API for understanding side topologies in STK topologies. Note that shell topologies with `shell_sides` have heterogenous side topologies that consist of `num_faces()` count of faces and `shell_sides`.

**Listing 3.15 Example for understanding side_topology in STK topology**
**code/stk/stk_doc_tests/stk_topology/shell_sides.cpp**

```
TEST(stk_topology, shell_side_topology) {
  stk::topology shell = stk::topology::SHELL_QUAD_4;

  EXPECT_TRUE(shell.is_valid());
  EXPECT_TRUE(shell.is_shell());

  EXPECT_EQ(shell.num_faces(),2u);
  EXPECT_EQ(shell.face_topology(0), stk::topology::QUAD_4);
  EXPECT_EQ(shell.face_topology(1), stk::topology::QUAD_4);

  EXPECT_EQ(shell.num_sides(),6u);
  EXPECT_EQ(shell.side_topology(0), stk::topology::QUAD_4);
  EXPECT_EQ(shell.side_topology(1), stk::topology::QUAD_4);
  EXPECT_EQ(shell.side_topology(2), stk::topology::SHELL_SIDE_BEAM_2);
  EXPECT_EQ(shell.side_topology(3), stk::topology::SHELL_SIDE_BEAM_2);
  EXPECT_EQ(shell.side_topology(4), stk::topology::SHELL_SIDE_BEAM_2);
  EXPECT_EQ(shell.side_topology(5), stk::topology::SHELL_SIDE_BEAM_2);
}
```

### 3.1.15. STK topology for a SuperElement

Listing 3.16 demonstrates the API for using super elements in STK Topology.

**Listing 3.16 Example using a SuperElement with STK topology**
**code/stk/stk_doc_tests/stk_topology/how_to_use_stk_topology.cpp**

```
TEST(stk_topology_understanding, superelements)
{
  unsigned eightNodes=8;
  stk::topology validSuperElement = stk::create_superelement_topology(eightNodes);
  EXPECT_TRUE(validSuperElement.is_superelement());
  EXPECT_TRUE(stk::topology::ELEMENT_RANK == validSuperElement.rank());
  EXPECT_EQ(eightNodes, validSuperElement.num_nodes());
  EXPECT_EQ(0u, validSuperElement.num_edges());
```

```
  EXPECT_EQ(0u, validSuperElement.num_faces());
  EXPECT_EQ(0u, validSuperElement.num_permutations());
  EXPECT_EQ(0u, validSuperElement.num_sides());
  EXPECT_EQ(0u, validSuperElement.dimension());
  EXPECT_EQ(stk::topology::INVALID_TOPOLOGY, validSuperElement.face_topology(0));
  EXPECT_EQ(stk::topology::INVALID_TOPOLOGY, validSuperElement.edge_topology(0));
  EXPECT_EQ(stk::topology::INVALID_TOPOLOGY, validSuperElement.base());
  EXPECT_FALSE(validSuperElement.has_homogeneous_faces());
  EXPECT_FALSE(validSuperElement.is_shell());

  unsigned zeroNodes=0;
  stk::topology invalidSuperElement = stk::create_superelement_topology(zeroNodes);
  EXPECT_FALSE(invalidSuperElement.is_superelement());
  EXPECT_TRUE(stk::topology::INVALID_RANK == invalidSuperElement.rank());
  EXPECT_EQ(zeroNodes, invalidSuperElement.num_nodes());
  EXPECT_EQ(0u, invalidSuperElement.num_edges());
  EXPECT_EQ(0u, invalidSuperElement.num_faces());
  EXPECT_EQ(0u, invalidSuperElement.num_permutations());
  EXPECT_EQ(0u, invalidSuperElement.num_sides());
  EXPECT_EQ(0u, invalidSuperElement.dimension());
  EXPECT_EQ(stk::topology::INVALID_TOPOLOGY, invalidSuperElement.face_topology(0));
  EXPECT_EQ(stk::topology::INVALID_TOPOLOGY, invalidSuperElement.edge_topology(0));
  EXPECT_EQ(stk::topology::INVALID_TOPOLOGY, invalidSuperElement.base());
  EXPECT_FALSE(invalidSuperElement.has_homogeneous_faces());
  EXPECT_FALSE(invalidSuperElement.is_shell());
}
```

## 3.2. Mapping of Sierra topologies

Listing 3.17 compares four topology implementations found in Sierra: the Exodus Topology (defined by the name and number of nodes of the element), Ioss Topology, STK Topology, and the Cell (Shards) Topology. The test shows how a few elements compare for these implementations.

**Listing 3.17 Example for understanding various Sierra topologies**
**code/stk/stk_doc_tests/stk_topology/understanding_various_topologies.cpp**

```

void setUpMappingsToTest(std::vector<TopologyMapper>& topologyMappings)
{
  std::string exodusName;
  int exodusNumNodes=-1;
  std::string iossTopologyName;
  stk::topology stkTopology;

  exodusName="sphere";
  exodusNumNodes=1;
  iossTopologyName="sphere";
  stkTopology=stk::topology::PARTICLE;
  topologyMappings.push_back(TopologyMapper(exodusName, exodusNumNodes, iossTopologyName,
            stkTopology));

  exodusName="BEam";
  exodusNumNodes=3;
  iossTopologyName="bar3";
  stkTopology=stk::topology::BEAM_3;
  topologyMappings.push_back(TopologyMapper(exodusName, exodusNumNodes, iossTopologyName,
            stkTopology));

  exodusName="Tri";
  exodusNumNodes=3;
  iossTopologyName="trishell3";
```

```
  stkTopology=stk::topology::SHELL_TRIANGLE_3;
  topologyMappings.push_back(TopologyMapper(exodusName, exodusNumNodes, iossTopologyName,
            stkTopology));

  exodusName="hex";
  exodusNumNodes=20;
  iossTopologyName="hex20";
  stkTopology=stk::topology::HEXAHEDRON_20;
  topologyMappings.push_back(TopologyMapper(exodusName, exodusNumNodes, iossTopologyName,
            stkTopology));

  exodusName="quad";
  exodusNumNodes=6;
  iossTopologyName="quad6";
  stkTopology=stk::topology::QUAD_6;
  topologyMappings.push_back(TopologyMapper(exodusName, exodusNumNodes, iossTopologyName,
            stkTopology));

  exodusName="wedge";
  exodusNumNodes=12;
  iossTopologyName="wedge12";
  stkTopology=stk::topology::WEDGE_12;
  topologyMappings.push_back(TopologyMapper(exodusName, exodusNumNodes, iossTopologyName,
            stkTopology));
}

TEST(Understanding, sierra_topologies)
{
  int spatialDim = 3;
  std::vector<TopologyMapper> topologyMappings;
  setUpMappingsToTest(topologyMappings);

  size_t numMappings = topologyMappings.size();

  createIossElementRegistryForKnownElementTopologies();

  for (size_t i=0;i<numMappings;i++)
  {
    TopologyMapper goldValues = topologyMappings[i];

    std::string fixedExodusName = Ioss::Utils::fixup_type(topologyMappings[i].exodusName,
            topologyMappings[i].exodusNumNodes, spatialDim);
    Ioss::ElementTopology *iossTopology = Ioss::ElementTopology::factory(fixedExodusName,
            true);
    ASSERT_TRUE(iossTopology != NULL);
    EXPECT_EQ(goldValues.iossTopologyName, iossTopology->name());

    stk::topology mappedStkTopologyFromIossTopology =
            stk::io::map_ioss_topology_to_stk(iossTopology, spatialDim);
    EXPECT_EQ(goldValues.stkTopology, mappedStkTopologyFromIossTopology);
  }
}
```

Some client applications still heavily use shards topologies with STK Mesh. To maintain support for this capability, STK Mesh provides a fast mapping between shards and stk_topology (see listing 3.18).

**Listing 3.18 Mapping of shards::CellTopologies to stk::topologies provided by stk::mesh::get_cell_topology()**
**code/stk/stk_mesh/stk_mesh/base/MetaData.cpp**

```
shards::CellTopology get_cell_topology(stk::topology t)
{
  switch(t())
  {
```

```
case stk::topology::NODE:
  return shards::CellTopology(shards::getCellTopologyData<shards::Node>());
case stk::topology::LINE_2:
  return shards::CellTopology(shards::getCellTopologyData<shards::Line<2>>());
case stk::topology::LINE_3:
  return shards::CellTopology(shards::getCellTopologyData<shards::Line<3>>());
case stk::topology::TRI_3:
  return shards::CellTopology(shards::getCellTopologyData<shards::Triangle<3>>());
case stk::topology::TRI_4:
  return shards::CellTopology(shards::getCellTopologyData<shards::Triangle<4>>());
case stk::topology::TRI_6:
  return shards::CellTopology(shards::getCellTopologyData<shards::Triangle<6>>());
case stk::topology::QUAD_4:
  return shards::CellTopology(shards::getCellTopologyData<shards::Quadrilateral<4>>());
case stk::topology::QUAD_6: break;
  //NOTE: shards does not define a topology for a 6-noded quadrilateral element
  // return shards::CellTopology(shards::getCellTopologyData<shards::Quadrilateral<6>>());
case stk::topology::QUAD_8:
  return shards::CellTopology(shards::getCellTopologyData<shards::Quadrilateral<8>>());
case stk::topology::QUAD_9:
  return shards::CellTopology(shards::getCellTopologyData<shards::Quadrilateral<9>>());
case stk::topology::PARTICLE:
  return shards::CellTopology(shards::getCellTopologyData<shards::Particle>());
case stk::topology::LINE_2_1D:
  return shards::CellTopology(shards::getCellTopologyData<shards::Line<2>>());
case stk::topology::LINE_3_1D:
  return shards::CellTopology(shards::getCellTopologyData<shards::Line<3>>());
case stk::topology::BEAM_2:
  return shards::CellTopology(shards::getCellTopologyData<shards::Beam<2>>());
case stk::topology::BEAM_3:
  return shards::CellTopology(shards::getCellTopologyData<shards::Beam<3>>());
case stk::topology::SHELL_LINE_2:
  return shards::CellTopology(shards::getCellTopologyData<shards::ShellLine<2>>());
case stk::topology::SHELL_LINE_3:
  return shards::CellTopology(shards::getCellTopologyData<shards::ShellLine<3>>());
case stk::topology::SPRING_2: break;
  //NOTE: shards does not define a topology for a 2-noded spring element
  //return shards::CellTopology(shards::getCellTopologyData<shards::Spring<2>>());
case stk::topology::SPRING_3: break;
  //NOTE: shards does not define a topology for a 3-noded spring element
  //return shards::CellTopology(shards::getCellTopologyData<shards::Spring<3>>());
case stk::topology::TRI_3_2D:
  return shards::CellTopology(shards::getCellTopologyData<shards::Triangle<3>>());
case stk::topology::TRI_4_2D:
  return shards::CellTopology(shards::getCellTopologyData<shards::Triangle<4>>());
case stk::topology::TRI_6_2D:
  return shards::CellTopology(shards::getCellTopologyData<shards::Triangle<6>>());
case stk::topology::QUAD_4_2D:
  return shards::CellTopology(shards::getCellTopologyData<shards::Quadrilateral<4>>());
case stk::topology::QUAD_8_2D:
  return shards::CellTopology(shards::getCellTopologyData<shards::Quadrilateral<8>>());
case stk::topology::QUAD_9_2D:
  return shards::CellTopology(shards::getCellTopologyData<shards::Quadrilateral<9>>());
case stk::topology::SHELL_TRI_3:
  return shards::CellTopology(shards::getCellTopologyData<shards::ShellTriangle<3>>());
case stk::topology::SHELL_TRI_4: break;
  //NOTE: shards does not define a topology for a 4-noded triangular shell
  //return shards::CellTopology(shards::getCellTopologyData<shards::ShellTriangle<4>>());
case stk::topology::SHELL_TRI_6:
  return shards::CellTopology(shards::getCellTopologyData<shards::ShellTriangle<6>>());
case stk::topology::SHELL_QUAD_4:
  return shards::CellTopology(shards::getCellTopologyData<shards::ShellQuadrilateral<4>>());
case stk::topology::SHELL_QUAD_8:
  return shards::CellTopology(shards::getCellTopologyData<shards::ShellQuadrilateral<8>>());
case stk::topology::SHELL_QUAD_9:
  return shards::CellTopology(shards::getCellTopologyData<shards::ShellQuadrilateral<9>>());
case stk::topology::TET_4:
  return shards::CellTopology(shards::getCellTopologyData<shards::Tetrahedron<4>>());
```

```
  case stk::topology::TET_8:
    return shards::CellTopology(shards::getCellTopologyData<shards::Tetrahedron<8>>());
  case stk::topology::TET_10:
    return shards::CellTopology(shards::getCellTopologyData<shards::Tetrahedron<10>>());
  case stk::topology::TET_11:
    return shards::CellTopology(shards::getCellTopologyData<shards::Tetrahedron<11>>());
  case stk::topology::PYRAMID_5:
    return shards::CellTopology(shards::getCellTopologyData<shards::Pyramid<5>>());
  case stk::topology::PYRAMID_13:
    return shards::CellTopology(shards::getCellTopologyData<shards::Pyramid<13>>());
  case stk::topology::PYRAMID_14:
    return shards::CellTopology(shards::getCellTopologyData<shards::Pyramid<14>>());
  case stk::topology::WEDGE_6:
    return shards::CellTopology(shards::getCellTopologyData<shards::Wedge<6>>());
  case stk::topology::WEDGE_12: break;
    //NOTE: shards does not define a topology for a 12-noded wedge
    // return shards::CellTopology(shards::getCellTopologyData<shards::Wedge<12>>());
  case stk::topology::WEDGE_15:
    return shards::CellTopology(shards::getCellTopologyData<shards::Wedge<15>>());
  case stk::topology::WEDGE_18:
    return shards::CellTopology(shards::getCellTopologyData<shards::Wedge<18>>());
  case stk::topology::HEX_8:
    return shards::CellTopology(shards::getCellTopologyData<shards::Hexahedron<8>>());
  case stk::topology::HEX_20:
    return shards::CellTopology(shards::getCellTopologyData<shards::Hexahedron<20>>());
  case stk::topology::HEX_27:
    return shards::CellTopology(shards::getCellTopologyData<shards::Hexahedron<27>>());
  default: break;
  }
  return shards::CellTopology(NULL);
}
```

# 4. STK MESH

## 4.1. STK Mesh Terms

Note that the concepts that define STK Mesh have been documented in some detail in [3]. A *Mesh* is a collection of *entities*, *parts*, *fields*, and *field data*. The STK Mesh API separates these collections into *MetaData* and *BulkData*.

Each of these terms is defined below.

### 4.1.1. Entity

*Entity* is a general term for the following types (listed in ascending 'rank' order): node, edge, face, element, and constraint. *Rank* is an enumerated type that describes and orders the different kinds of entities.

### 4.1.2. Connectivity

In a finite element discretization, entities are connected to other entities. Examples include: element-to-node connectivity (the nodes connected to a given element), node-to-element connectivity (the elements connected to a given node), and face-to-element connectivity (the elements connected to a given face). A connection from a higher-rank entity to a lower-rank entity is referred to as a *downward relation*. When a downward relation is declared (e.g., between an element and a node), STK Mesh, by default, creates the corresponding *upward relation* (e.g., from the node to the element). Table 4-1 shows the default connectivity of a fully-connected mesh. The term fully-connected means that the client code has established all downward relations. The term fixed means that the number of relations is defined by topology; the number of node-relations for a hex-8 element is 8. The term dynamic means that the number of relations is unknown until individual relations have been established. For example, an element may have 0, 1, or more faces depending on whether it is on an external boundary. STK mesh provides functions for creating all edges or faces (see Sections 4.6.7 and 4.6.8). It should be noted that STK Mesh does not support connectivity between entities of the same rank. As an additional note, the term *relations* and *connectivity* are used interchangeably in this document.

### 4.1.3. Topology

*Topology* provides an entity's finite element description. This includes several attributes such as the number and type of lower-rank entities that can exist in that entity's downward relations. For

**Table 4-1. Default connectivity of a fully-connected mesh**

| From-entity | To Node | To Edge | To Face | To Element |
|---|---|---|---|---|
| Node | - | dynamic | dynamic | dynamic |
| Edge | fixed | - | dynamic | dynamic |
| Face | fixed | dynamic | - | dynamic |
| Element | fixed | dynamic | dynamic | - |

example, an element with hex8 topology must have 8 nodes and may have up to a maximum of 6 quad4 faces and 12 line2 edges. Quad4, line2, and nodes are also examples of topologies. Topology also defines what *permutations* in downward connectivity are permissible. Unlike downward connectivity, upward connectivity is determined at run-time and does not imply restrictions on permutations. See chapter 3 for more detail about the STK Topology component and examples of using the API.

Note that in STK Mesh, entities with entity-rank higher than element-rank generally don't have an associated topology.

#### 4.1.4. Part

*Part* is a general term for a subset of entities in a mesh. Parts are a grouping mechanism used to operate on subsets of the mesh (see Section 4.1.6). STK Mesh automatically creates four parts at startup: the *universal part*, the *locally-owned part*, the *globally-shared part*, and the *aura part*. These parts are important to the basic understanding of ghosting (see Section 4.1.8). For meshes read from Exodus files, additional Exodus parts are created (blocks, sidesets, and nodesets). Each entity in the mesh must be a member of one or more parts.

Parts exist for the life of the STK Mesh; parts cannot be deleted without deleting the mesh. STK Mesh provides methods which allow client code to explicitly change the user-defined part membership of an entity.

See Section 4.4 for more details on mesh parts.

#### 4.1.5. Field

*Fields* are data associated with mesh entities. Examples include coordinates, velocity, displacement, and temperature. A field in STK Mesh can hold any data type (e.g., double or int) and any number of scalars per entity (e.g., nodal velocity field has three doubles per node if the spatial dimension is 3). A field can be allocated (defined) on the whole mesh (e.g., all nodes) or on a Part (subset) of the mesh (nodes of a sideset). For example, a material property can be defined on a specified element block.

#### 4.1.6. Selector

*Selectors* are used to select entities that belong to a specified expression of parts. Here are some examples:

- Select all elements that are in either block-1 or block-2 or both. (A set-union expression.)
- Select all nodes that are connected to elements in both block-1 and block-2. (A set-intersection expression.)
- Select all nodes that are locally-owned but not connected to a rigid-body part. (A set-difference expression.)
- Select all nodes that have a specified field allocated. Since field allocation is specified in terms of parts, we allow selectors to be created based on fields.

The selector system is explained further in Section 4.5.

### 4.1.7. Bucket

STK Mesh organizes entities into *buckets*: the entities in a bucket all have the same rank and topology, and they are all members of the same parts. Additionally, the entities in a bucket correspond to contiguously-allocated blocks of memory in the associated field-data values.

There are two primary reasons for grouping entities into buckets. Firstly, the Selector system (see section 4.5) allows for the traversal of the mesh in arbitrary user-defined subsets, and these subsets exist as combinations of buckets. Secondly, the performance of mesh-modification (see section 4.6) is improved by only moving bucket-sized sections of allocated memory (e.g., when adding/deleting entities) rather than re-allocating and sliding the memory for the whole mesh.

No entity is ever in more than one bucket at any given time. This grouping is performed internally by STK Mesh; client code has no explicit control over which entities reside in which buckets. If an entity's part membership is changed, it is automatically moved to a different bucket.

### 4.1.8. Ghosting

*Ghosting* in STK Mesh provides a way to perform operations that involve entities that are neither locally-owned nor shared on the current processor. STK Mesh automatically provides a one-element thick ghost layer around each processor, referred to as the *aura* and is shown in Figures 4-1 and 4-2. Formally, the aura is defined as a ghosting of the upward-relations for shared entities. In other words, if the aura is on, then shared entities have the same upward-relations on each sharing processor. In addition, STK Mesh client code can also request arbitrary ghosting of entities, referred to as custom ghosting.

### 4.1.9. MetaData and BulkData

The *MetaData* component of a STK Mesh contains the definitions of its parts, the definitions of its fields, and definitions of relationships among its parts and fields. For example, a subset relationship can be declared between two parts, and a field definition can be limited to specific parts. The *BulkData* component of a STK Mesh contains entities, entity ownership and ghosting

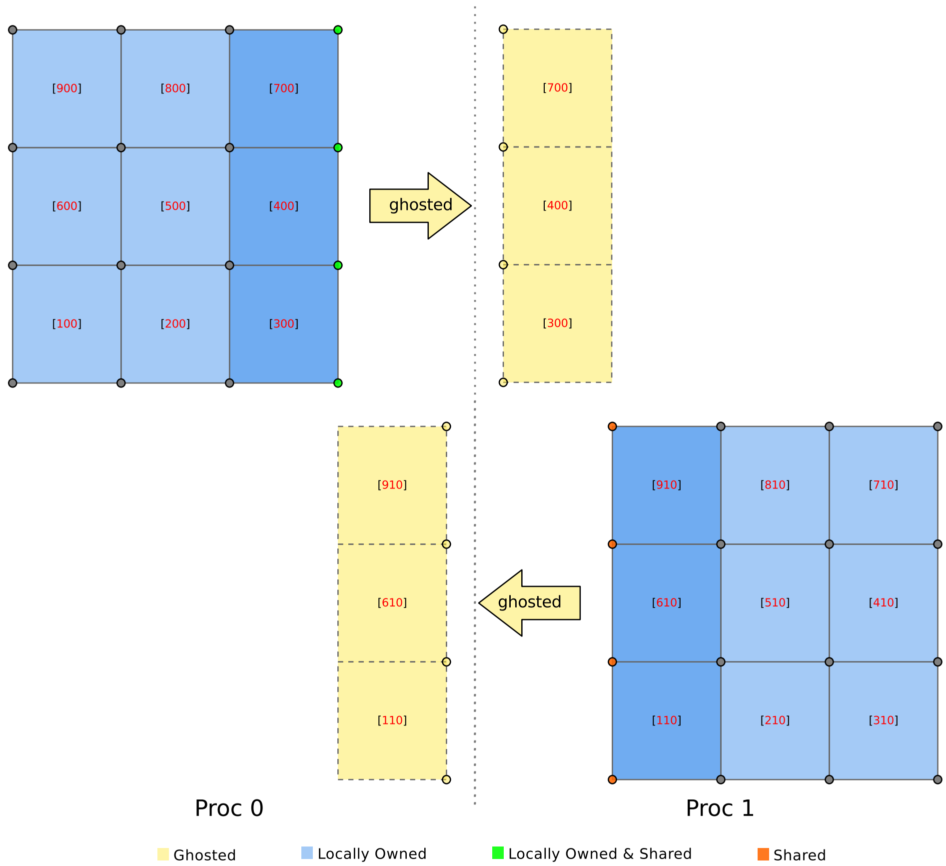



**Figure 4-1. Aura ghosting per MPI process**

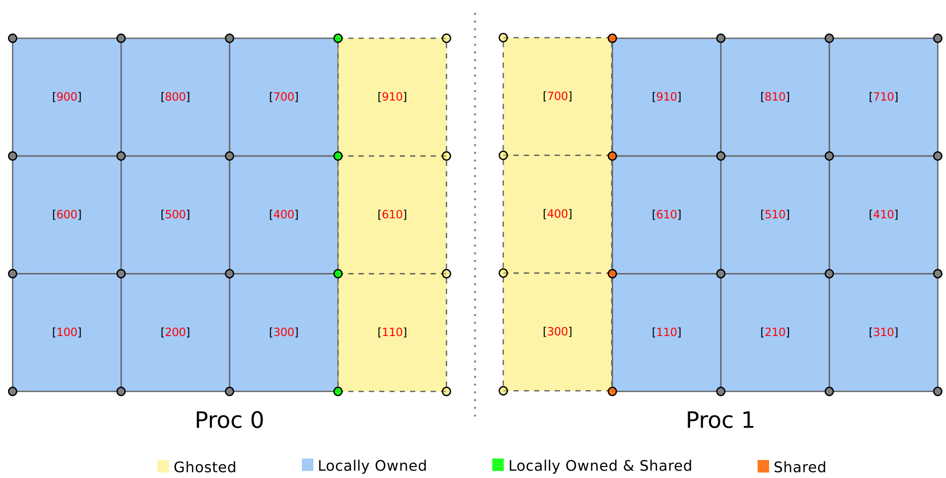



**Figure 4-2. Final auras**

information, connectivity data, and field data. For efficiency, the BulkData API enables data access via buckets, in addition to data access via entity and rank.

A mesh's MetaData holds a database definition (a schema), and a mesh's BulkData holds the content of that database. MetaData is replicated (duplicated) on all processors; BulkData is distributed across processors with each processor having a separate subset of the data, subject to sharing and ghosting.

This design requires object construction of MetaData and BulkData to be staged. The spatial dimension of a mesh is usually specified in the call to the MetaData constructor, which also provides a valid default initialization. The BulkData constructor requires a MetaData object as an argument. A BulkData object cannot be modified (e.g., entities added) before its MetaData object has been initialized and then committed using the `MetaData::commit()` member function (for example, see Listing 3.1). Once a MetaData object has been committed, it cannot be changed. Therefore, fields must be put on parts prior to MetaData commit. Non-topology parts can still be declared after commit, but they will have limited uses because subset relationships cannot be changed. For clarity, it is recommended that MetaData commit is called prior to BulkData construction. If **new** is used to create a BulkData object, then that instance must be deleted before its MetaData object (used to construct it) is destroyed.

The STK Mesh usage examples below and in Section 4.7 illustrate common uses of the MetaData and BulkData APIs.

#### *4.1.10. Creating a STK Mesh from an Exodus file*

Listing 4.1 shows how to create and populate a STK Mesh using the STK IO module, which is described in Chapter 5. We provide this example for those who want to quickly get started using an STK Mesh given an Exodus file. This particular example shows STK IO populating the STK Mesh from a generated-in-memory mesh, but the “filename” is all that would need to change, to instead read data from an Exodus file. Further examples will show various uses of the STK Mesh.

**Listing 4.1 Example of creating an STK Mesh using an Exodus file**
**code/stk/stk_doc_tests/stk_mesh/createStkMesh.cpp**

```
TEST(StkMeshHowTo, UseStkIO)
{
  MPI_Comm communicator = MPI_COMM_WORLD;
  if(stk::parallel_machine_size(communicator) == 1)
  {
    std::shared_ptr<stk::mesh::BulkData> bulkPtr =
             stk::mesh::MeshBuilder(communicator).create();

    stk::io::StkMeshIoBroker meshReader;
    meshReader.set_bulk_data(*bulkPtr);
    meshReader.add_mesh_database("generated:8x8x8", stk::io::READ_MESH);
    meshReader.create_input_mesh();
    meshReader.add_all_mesh_fields_as_input_fields();
    meshReader.populate_bulk_data();

    unsigned numElems = stk::mesh::count_entities(*bulkPtr, stk::topology::ELEM_RANK,
             bulkPtr->mesh_meta_data().universal_part());
    EXPECT_EQ(512u, numElems);
  }
}
```

After these steps, the STK Mesh objects now contain all the data from the Exodus file (e.g., Fields, Parts, Entities).

## 4.2. Parallel

STK Mesh maintains a parallel consistent mesh across many MPI processes or subdomains. Most of the parallel capabilities revolve around communicating information, like field data, for entities on the boundaries of these subdomains. Entities that are communicated between subdomains are either shared or ghosted.

### *4.2.1. Shared*

Entities that are shared among processors are downward connected from a locally-owned entity, usually an element. For example, if the side of a hex8 is on a subdomain boundary, the 4 nodes that touch the boundary are considered *shared*. If there also exists a face on that side of the hex, the face would also be shared.

Shared entities have fully symmetric communication information stored on all processors that share the entity. In other words, every processor that has a shared entity knows about every other processor that shares the entity.

### *4.2.2. Ghosted*

*Ghosted* entities are communicated between subdomains regardless of the connections from locally-owned entities. This is different from shared entities which are defined by downward connection from locally-owned entities.

Ghosted entities only have communication information about the owner stored on the processor that the entities are ghosted to. This means that a given processor's BulkData has information about the processor the ghost came from but not any other processors that the entity may have been ghosted to.

### *4.2.3. Aura*

The *aura* is a special ghosting that automatically sends one layer of ghosted elements on the subdomain boundaries to the processors that share those boundaries, as seen in Figures 4-1 and 4-2. The aura can be turned off when the mesh is initially created. See Section 4.2.3.1 for example usage.

#### 4.2.3.1. How to use automatically generated aura

This section describes how to control whether or not a one-layer ghosting of elements is automatically generated around each processor's mesh.

**Listing 4.2 Example of how to control automatically generated aura**
**code/stk/stk_doc_tests/stk_mesh/howToUseAura.cpp**

```
void expectNumElementsInAura(stk::mesh::BulkData::AutomaticAuraOption autoAuraOption,
```

```
                              unsigned numExpectedElementsInAura)
{
  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) == 2)
  {
    stk::mesh::MeshBuilder builder(MPI_COMM_WORLD);
    builder.set_aura_option(autoAuraOption);
    std::shared_ptr<stk::mesh::BulkData> bulkPtr = builder.create();
    stk::mesh::MetaData& meta = bulkPtr->mesh_meta_data();
    stk::mesh::BulkData& bulk = *bulkPtr;
    stk::io::fill_mesh("generated:1x1x2", bulk);

    EXPECT_EQ(numExpectedElementsInAura, stk::mesh::count_entities(bulk,
             stk::topology::ELEM_RANK, meta.aura_part()));
  }
}
TEST(StkMeshHowTo, useNoAura)
{
  expectNumElementsInAura(stk::mesh::BulkData::NO_AUTO_AURA, 0);
}
TEST(StkMeshHowTo, useAutomaticGeneratedAura)
{
  expectNumElementsInAura(stk::mesh::BulkData::AUTO_AURA, 1);
}
```

## 4.3. STK Parallel Mesh Consistency Rules

STK Mesh is used by many engineering disciplines such as structural dynamics, solid mechanics, thermal/fluid mechanics, and mesh refinement. Since the mesh is being used by different applications, we must ensure that the mesh is **consistent**. A consistent mesh will always follow certain rules/guidelines regardless of the application using it. This has a disadvantage in that flexibility to tune/adjust the mesh for a specific application's needs is reduced, but it also allows easier coupling between applications and helps reuse of algorithms that use STK Mesh because of these rules.

Much of the work in STK Mesh, during modification cycles, is towards creating a consistent mesh especially in parallel. The following are some of the ideas behind a parallel consistent mesh:

- For entities with the same identifier (EntityKey), then for all the processors that have the entity
  - the owner is the same
  - the application-defined parts that the entity is a member of, are the same
  - every entity has the same downward relations on all processors
  - every entity has the same upward relations on all processors (only if the aura is active)
- For aura'ed/shared entities
  - owner of entity knows with which processors the entity is shared with and/or aura'ed to
  - sharer (not owner) of entity knows which other processors share the entity
  - processor with aura'ed entity knows the owner of the entity

At first glance, these rules might seem trivial. The STK Mesh API prevents the ability to change mesh to get it into an inconsistent state at the end of a modification cycle. This concept has proven to be powerful in that it allows coupling of codes and reuse of algorithms across applications.

### *4.3.1. How to enable mesh diagnostics to enforce parallel mesh rules*

STK Mesh provides a means by which an application may enable internal mesh diagnostics to ensure that the mesh is consistent with the three Parallel Mesh Rules (PMR). These rules may be summarized as:

- Rule 1: Coincident and partially coincident elements must be owned by the same processor (no split coincident elements)
- Rule 2: Each global id shall be owned by one and only one processor (no duplicate ids)
- Rule 3: Processor that owns a side also owns at least one element to which it is connected. (each side needs an element i.e no solo faces)

Enabling mesh diagnostics creates a per-processor file named “mesh_diagnostics_failures_<proc_id>.txt” which contains the listing of all errors. This example demonstrates first creating a mesh with a sideset and then checking that there are no solo faces with attached elements that are remotely owned (PMR-3).

**Listing 4.3 Example of how to enable mesh diagnostics**
**code/stk/stk_doc_tests/stk_mesh/howToEnableMeshDiagnostics.cpp**

```
TEST(StkMeshHowTo, EnableMeshDiagnostics)
{
  std::shared_ptr<stk::mesh::BulkData> bulkPtr =
            stk::mesh::MeshBuilder(MPI_COMM_WORLD).create();
  stk::io::fill_mesh("generated:4x4x4|sideset:xX", *bulkPtr);

  bulkPtr->enable_mesh_diagnostic_rule(stk::mesh::RULE_3);
  EXPECT_EQ(0u, bulkPtr->get_mesh_diagnostic_error_count());
}
```

### *4.3.2. How to enforce Parallel Mesh Rule 1*

STK Mesh provides a means by which an application may enforce Parallel Mesh Rule 1 (PMR-1) to ensure that coincident and partially-coincident elements must be owned by the same processor (no split coincident elements).

**Listing 4.4 Example of how to enforce Parallel Mesh Rule 1**
**code/stk/stk_doc_tests/stk_balance/howToFixPMR1Violation.cpp**

```
TEST(StkMeshHowTo, FixPMR1Violation)
{
    stk::mesh::MetaData meta;
    std::shared_ptr<stk::mesh::BulkData> bulkData =
            stk::mesh::MeshBuilder(MPI_COMM_WORLD).create();
    stk::io::fill_mesh("generated:4x4x4|sideset:xX", *bulkData);

    stk::mesh::EntityIdProcMap elementAndDestProc;
```

```
    EXPECT_NO_THROW(elementAndDestProc =
              stk::balance::make_mesh_consistent_with_parallel_mesh_rule1(*bulkData));
    EXPECT_TRUE(elementAndDestProc.size()==0u); // no elements were migrated
}
```

#### 4.3.3. Parallel API

This section discusses a few API functions for applications using the parallel capabilities of STK Mesh.

The following code example shows how to communicate field data from owned to all shared and ghosted entities, overwriting any local modifications.

**Listing 4.5 Example of communicating field data from owned to all shared and ghosted entities**
**code/stk/stk_doc_tests/stk_mesh/communicateFieldData.cpp**

```
TEST_F(ParallelHowTo, communicateFieldDataForSharedAndAura)
{
  setup_empty_mesh(stk::mesh::BulkData::AUTO_AURA);
  auto& field = get_meta().declare_field<double>(stk::topology::NODE_RANK, "temperature");

  double initialValue = 25.0;
  stk::mesh::put_field_on_entire_mesh_with_initial_value(field, &initialValue);

  stk::io::fill_mesh("generated:8x8x8", get_bulk());

  stk::mesh::Selector notOwned = !get_meta().locally_owned_part();
  stk::mesh::for_each_entity_run(get_bulk(), stk::topology::NODE_RANK, notOwned,
    [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
      *stk::mesh::field_data(field, node) = -1.2345;
    });

  stk::mesh::communicate_field_data(get_bulk(), {&field});

  stk::mesh::for_each_entity_run(get_bulk(), stk::topology::NODE_RANK, notOwned,
    [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
      EXPECT_EQ(initialValue, *stk::mesh::field_data(field, node));
    });
}
```

The parallel_sum, parallel_min, and parallel_max functions operate on shared entities.

**Listing 4.6 Example of parallel_sum**
**code/stk/stk_doc_tests/stk_mesh/communicateFieldData.cpp**

```
void expect_nodal_field_has_value(const stk::mesh::Selector& selector,
                                  const stk::mesh::Field<double> &field,
                                  const double value)
{
  stk::mesh::for_each_entity_run(field.get_mesh(), stk::topology::NODE_RANK, selector,
    [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
      EXPECT_EQ(value, *stk::mesh::field_data(field, node));
    });
}

TEST_F(ParallelHowTo, computeParallelSum)
{
  setup_empty_mesh(stk::mesh::BulkData::AUTO_AURA);
  auto& field = get_meta().declare_field<double>(stk::topology::NODE_RANK, "temperature");

```

```
  double initialValue = 25.0;
  stk::mesh::put_field_on_entire_mesh_with_initial_value(field, &initialValue);

  stk::io::fill_mesh("generated:8x8x8", get_bulk());

  expect_nodal_field_has_value(get_meta().globally_shared_part(), field, initialValue);
  expect_nodal_field_has_value(!get_meta().globally_shared_part(), field, initialValue);

  stk::mesh::parallel_sum(get_bulk(), {&field});

  expect_nodal_field_has_value(get_meta().globally_shared_part(), field, 2*initialValue);
  expect_nodal_field_has_value(!get_meta().globally_shared_part(), field, initialValue);
}
```

The `comm_mesh_counts` function is shown in Listings 4.7-4.8. The purpose of this function is to count the number of entities of each entity rank across all processors.

**Listing 4.7 Example showing parallel use of comm_mesh_counts**
**code/stk/stk_doc_tests/stk_mesh/UnitTestCommMeshCounts.cpp**

```
TEST( CommMeshCounts, Parallel )
{
    stk::ParallelMachine communicator = MPI_COMM_WORLD;
    int numprocs = stk::parallel_machine_size(communicator);

    const std::string generatedMeshSpec = getGeneratedMeshString(10,20,2*numprocs);
    stk::unit_test_util::StkMeshCreator stkMesh(generatedMeshSpec, communicator);

    std::vector<size_t> comm_mesh_counts;
    stk::mesh::comm_mesh_counts(*stkMesh.getBulkData(), comm_mesh_counts);

    size_t goldNumElements = 10*20*2*numprocs;
    EXPECT_EQ(goldNumElements, comm_mesh_counts[stk::topology::ELEMENT_RANK]);
}
```

**Listing 4.8 Example showing parallel use of comm_mesh_counts with min/max counts**
**code/stk/stk_doc_tests/stk_mesh/UnitTestCommMeshCounts.cpp**

```
TEST( CommMeshCountsWithStats, Parallel )
{
    stk::ParallelMachine communicator = MPI_COMM_WORLD;
    int numprocs = stk::parallel_machine_size(communicator);

    const std::string generatedMeshSpec = getGeneratedMeshString(10,20,2*numprocs);
    stk::unit_test_util::StkMeshCreator stkMesh(generatedMeshSpec, communicator);

    std::vector<size_t> comm_mesh_counts;
    std::vector<size_t> min_counts;
    std::vector<size_t> max_counts;

    stk::mesh::comm_mesh_counts(*stkMesh.getBulkData(), comm_mesh_counts, min_counts,
            max_counts);

    size_t goldNumElements = 10*20*2*numprocs;
    EXPECT_EQ(goldNumElements, comm_mesh_counts[stk::topology::ELEMENT_RANK]);

    size_t goldMinNumElements = 10*20*2;
    EXPECT_EQ(goldMinNumElements, min_counts[stk::topology::ELEMENT_RANK]);

    size_t goldMaxNumElements = goldMinNumElements;
    EXPECT_EQ(goldMaxNumElements, max_counts[stk::topology::ELEMENT_RANK]);
}
```

## 4.4. STK Mesh Parts

A *mesh part* is a subset of entities of the mesh, and may be used to reflect the physics modeled, discretization methodology, solution algorithm, meshing artifacts, or other application specific requirements.

STK Mesh automatically defines several parts during initialization, demonstrated here based on the serial The *universal part* includes every entity on the current MPI process (Figure 4-3). The *locally-owned part* contains all the entities owned by the current MPI process (Figure 4-4). The *globally-shared part* contains all the entities on the current MPI process that are shared with another MPI process, whether locally-owned or not. Figures 4-5 and 4-6 illustrate the globally shared part. An entity may be in both the locally-owned and globally-shared parts. By default, a shared entity is owned by the lowest-numbered sharing MPI process, though client code is allowed to change entity ownership. Part declarations and part membership are consistent across processor ranks; part membership for a given entity is maintained on the owning rank. The *aura part* contains all the entities which are ghosted due to aura. An additional part is kept up-to-date for each custom ghosting and examples of usage are in Section 4.4.3.

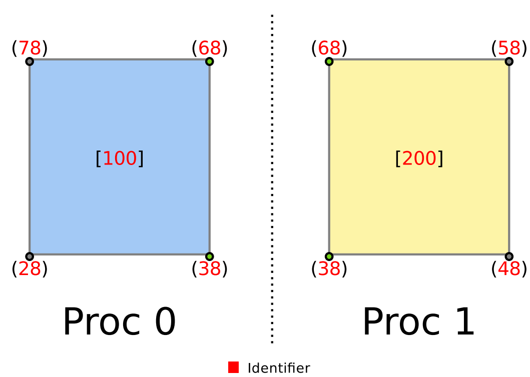



**Figure 4-3. Parallel-decomposed STK Mesh. This figure depicts the universal parts on each process.**

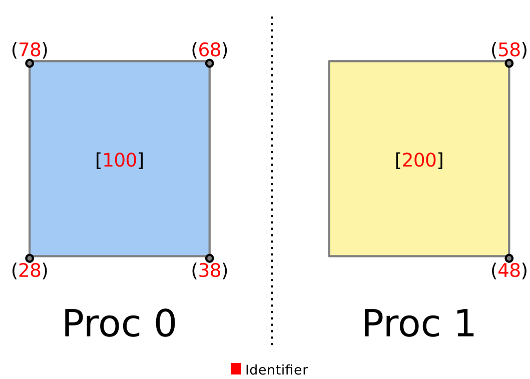



**Figure 4-4. Locally-owned parts. Nodes 38 and 68 are owned by process 1 and are not in process 2's locally-owned part.**

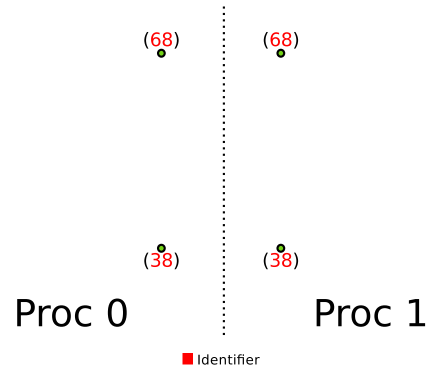



**Figure 4-5. Globally-shared parts. Nodes 38 and 68 appear in both process's globally-shared part.**

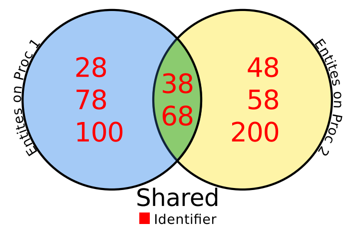



**Figure 4-6. Entities in the globally-shared part from each process.**

### *4.4.1. Part Identifiers and Attributes*

A mesh part has a unique text name identifier, specified by the application that creates the part. This identifier is intended to support text input and output by the application, e.g., parsing, logging, and error reporting. The text name is not intended for referencing a mesh part within application computations. As reliance on text-based references will lead to text-based searches within the application's computations, resulting in unnecessarily degraded performance.

A mesh part also has a unique non-negative integer identifier, its *part ordinal*, that is internally generated by the mesh MetaData. Part ordinals are intended to support fast referencing and ordering of mesh parts. The part ordinal is also intended to support efficient communication of mesh part information among distributed memory processes.

An application, for example, may specify a mesh part for an *element block* (a collection of elements); in descriptions of part behavior, we use the following notation:

$$\begin{aligned} Part_A &\equiv \text{mesh part identified by } A \\ Part_A^J &\equiv \text{mesh part intended for mesh entities of rank } J \text{ and identified by } A \end{aligned} \tag{4.1}$$

Note that all processors have the same part list. Hence, parts must be created synchronously across all processors to avoid part lists becoming different on any processor.

### 4.4.2. *Induced Part Membership*

An application can explicitly insert a mesh entity into a mesh part or explicitly remove a mesh entity from a part. A mesh entity's membership in a part may also be induced through its connectivity to a higher rank mesh entity. Thus, a mesh entity may be an *explicit member* or an *induced member* of a mesh part.

For example, a node will have induced membership in an element block (mesh part) when that node has connectivity from an element that is in that part. Therefore, the nodes of all the elements in the element block will be in that part due to induced part membership. This enables client code to select and iterate over the nodes of the elements in the element block directly and uniquely, rather than through element connectivity. In general, the explicit part membership of a given entity automatically induces the same part membership onto any lower-ranking entities that are connected to it.

When a mesh part has a specified entity rank ($Part_A^J$) then only mesh entities of the same entity rank $J$ may be explicitly added as members to that mesh part. If a mesh entity is an *explicit member* of such a mesh part, $entity_a^J \in Part_A^J$, and that mesh entity ($entity_a^J$) is the from-entity of a connectivity, then the to-entity of that connectivity is an *induced member* of that mesh part. More formally,

$$\begin{array}{lll} \text{Given} & \text{a connectivity } \left( entity_a^J \,,\, entity_b^K \,,\, x \right) \;:\; J > K \quad \text{and} & \\ & entity_a^J \in Part_A^J \quad \text{via explicit membership} & \\ \text{then} & entity_b^K \in Part_A^J \quad \text{via induced membership.} & \end{array} \tag{4.2}$$

Note that induced-part memberships are added (or removed) whenever a connectivity is declared (or deleted). As a result, declaring or deleting a connectivity can cause an entity to move to a different bucket.

Induced membership only occurs in the presence of a mesh entity connectivity. This means that induced membership is **not** transitive. For example, if a mesh has both element-to-face and face-to-edge connectivities, but does not have element-to-edge connectivities, then the edges in the element's closure (via element-to-face-to-edge) are **not** induced members.

### 4.4.3. *How to use ghost parts*

These examples demonstrate how to use the ghost parts to select those entities that are ghosted due to aura or custom ghosting.

**Listing 4.9 Example of how to use Ghost Parts to select aura ghosts and custom ghosts**
**code/stk/stk_doc_tests/stk_mesh/UnitTestGhostParts.cpp**

```
TEST(UnitTestGhostParts, Aura)
{
  stk::ParallelMachine communicator = MPI_COMM_WORLD;

  int numProcs = stk::parallel_machine_size(communicator);
  if (numProcs != 2) {
    return;
  }

```

```
  stk::io::StkMeshIoBroker stkMeshIoBroker(communicator);
  const std::string generatedMeshSpecification = "generated:1x1x3";
  stkMeshIoBroker.add_mesh_database(generatedMeshSpecification, stk::io::READ_MESH);
  stkMeshIoBroker.create_input_mesh();
  stkMeshIoBroker.populate_bulk_data();

  stk::mesh::MetaData &stkMeshMetaData = stkMeshIoBroker.meta_data();
  stk::mesh::BulkData &stkMeshBulkData = stkMeshIoBroker.bulk_data();

  std::cerr<<"about to get aura_part..."<<std::endl;
  stk::mesh::Part& aura_part = stkMeshMetaData.aura_part();
  std::cerr<<"...got aura part with name="<<aura_part.name()<<std::endl;
  stk::mesh::Selector aura_selector = aura_part;

  stk::mesh::Ghosting& aura_ghosting = stkMeshBulkData.aura_ghosting();
  EXPECT_EQ(aura_part.mesh_meta_data_ordinal(),
            stkMeshBulkData.ghosting_part(aura_ghosting).mesh_meta_data_ordinal());

  stk::mesh::Selector not_owned_nor_shared = (!stkMeshMetaData.locally_owned_part()) &
            (!stkMeshMetaData.globally_shared_part());

  const stk::mesh::BucketVector& not_owned_nor_shared_node_buckets =
            stkMeshBulkData.get_buckets(stk::topology::NODE_RANK, not_owned_nor_shared);
  size_t expected_num_not_owned_nor_shared_node_buckets = 1;
  EXPECT_EQ(expected_num_not_owned_nor_shared_node_buckets,
            not_owned_nor_shared_node_buckets.size());

  const stk::mesh::BucketVector& aura_node_buckets =
            stkMeshBulkData.get_buckets(stk::topology::NODE_RANK, aura_selector);

  EXPECT_EQ(not_owned_nor_shared_node_buckets.size(), aura_node_buckets.size());

  const size_t expected_num_ghost_nodes = 4;
  size_t counted_nodes = 0;
  size_t counted_aura_nodes = 0;
  for(size_t i=0; i<not_owned_nor_shared_node_buckets.size(); ++i)
  {
    counted_nodes += not_owned_nor_shared_node_buckets[i]->size();
    counted_aura_nodes += aura_node_buckets[i]->size();
  }
  EXPECT_EQ(expected_num_ghost_nodes, counted_nodes);
  EXPECT_EQ(expected_num_ghost_nodes, counted_aura_nodes);
}

TEST(UnitTestGhostParts, Custom1)
{
  stk::ParallelMachine communicator = MPI_COMM_WORLD;

  int numProcs = stk::parallel_machine_size(communicator);
  if (numProcs != 2) {
    return;
  }

  stk::io::StkMeshIoBroker stkMeshIoBroker(communicator);
  const std::string generatedMeshSpecification = "generated:1x1x4";
  stkMeshIoBroker.add_mesh_database(generatedMeshSpecification, stk::io::READ_MESH);
  stkMeshIoBroker.create_input_mesh();
  stkMeshIoBroker.populate_bulk_data();

  stk::mesh::BulkData &stkMeshBulkData = stkMeshIoBroker.bulk_data();

  int myProc = stkMeshBulkData.parallel_rank();
  int otherProc = (myProc == 0) ? 1 : 0;

  stkMeshBulkData.modification_begin();

  stk::mesh::Ghosting& custom_ghosting = stkMeshBulkData.create_ghosting("CustomGhosting1");

```

```
  std::vector<stk::mesh::EntityProc> elems_to_ghost;

  const stk::mesh::BucketVector& elem_buckets =
            stkMeshBulkData.buckets(stk::topology::ELEM_RANK);
  for(size_t i=0; i<elem_buckets.size(); ++i) {
    const stk::mesh::Bucket& bucket = *elem_buckets[i];
    for(size_t j=0; j<bucket.size(); ++j) {
      if (stkMeshBulkData.parallel_owner_rank(bucket[j]) == myProc) {
        elems_to_ghost.push_back(std::make_pair(bucket[j], otherProc));
      }
    }
  }

  stkMeshBulkData.change_ghosting(custom_ghosting, elems_to_ghost);

  stkMeshBulkData.modification_end();

  //now each processor should have 2 elements that were received as ghosts of elements from
            the other proc.
  const size_t expected_num_elems_for_custom_ghosting = 2;

  stk::mesh::Part& custom_ghost_part = stkMeshBulkData.ghosting_part(custom_ghosting);
  stk::mesh::Selector custom_ghost_selector = custom_ghost_part;

  const stk::mesh::BucketVector& custom_ghost_elem_buckets =
            stkMeshBulkData.get_buckets(stk::topology::ELEM_RANK, custom_ghost_selector);
  size_t counted_elements = 0;
  for(size_t i=0; i<custom_ghost_elem_buckets.size(); ++i) {
    counted_elements += custom_ghost_elem_buckets[i]->size();
  }

  EXPECT_EQ(expected_num_elems_for_custom_ghosting, counted_elements);
}
```

## 4.5. STK Mesh Selector

A *selector* is a set-logical expression, involving parts, that can include intersections, unions, and complements. The default-constructed selector is empty and therefore selects nothing. See Section 4.5.1 for examples.

A selector is typically used with `get_buckets()` for a given entity rank to get a list of buckets satisfying that selector. `get_buckets()` evaluates the selector on each bucket of the specified rank. When the expression evaluation gets down to a part, the selector must determine if that part is listed as one of the part intersections in the bucket. The worst-case cost of evaluating `get_buckets()` is

$$O\left(N_{number\ buckets}\right) \times O\left(N_{number\ selector\ terms}\right) \times O\left(N_{number\ bucket\ parts}\right) \tag{4.3}$$

where $N_{number\ buckets}$ is the number of buckets of the Entity rank that was passed into `get_buckets()`, $N_{number\ selector\ terms}$ is the length of the selector expression, and $N_{number\ bucket\ parts}$ is the average number of parts that each bucket represents.

Since STK Mesh internally caches the results of calls to `get_buckets()`, selector performance often does not have a large impact on overall application runtime. Selectors are implemented to allow optimization from short-circuiting logic, to allow a positive result from a union to ignore the rest of the expression, as well as a negative result from an intersection. If selectors are constructed to take advantage of this type of early termination, the middle term in equation (4.3) is less expensive in practice. For example, if `partA` strictly contains `partB`, then the selector expression `(partA | partB)` will tend to select more efficiently than `(partB | partA)` because, in the first case, once it is known that a bucket is selected for `partA`, that bucket does not need to be checked against `partB`.

### 4.5.1. How to use selectors

These examples demonstrate some basic creation and usage of Selectors, including performing set operations such as unions, intersections and differences. They also demonstrate retrieving the buckets associated with a Selector.

**Listing 4.10 Example of using Selectors, including the "Nothing" selector**
**code/stk/stk_doc_tests/stk_mesh/howToUseSelectors.cpp**

```
TEST(StkMeshHowTo, basicSelectorUsage)
{
  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) != 1) { GTEST_SKIP(); }
  std::unique_ptr<stk::mesh::BulkData> bulkPtr = stk::mesh::MeshBuilder(communicator)
                                                   .set_spatial_dimension(3).create();
  stk::mesh::MetaData& meta = bulkPtr->mesh_meta_data();

//create a simple shell-quad-4 mesh:
//        6
// 3*----*----*9
//  | E2 | E4 |
//  |    |    |
// 2*---5*----*8
//  | E1 | E3 |
//  |    |    |
// 1*----*----*7
//        4
//

  std::string meshDesc = "0,1,SHELL_QUAD_4, 1,4,2,5, block_1\n"
                         "0,2,SHELL_QUAD_4, 2,5,6,3, block_2\n"
                         "0,3,SHELL_QUAD_4, 4,7,8,5, block_3\n"
                         "0,4,SHELL_QUAD_4, 5,8,9,6, block_4\n";
  stk::unit_test_util::setup_text_mesh(*bulkPtr, meshDesc);

  stk::mesh::Part& block_1 = *meta.get_part("block_1");
  stk::mesh::Part& block_2 = *meta.get_part("block_2");
  stk::mesh::Part& block_3 = *meta.get_part("block_3");
  stk::mesh::Part& block_4 = *meta.get_part("block_4");
  stk::mesh::PartVector allBlocks = {&block_1, &block_2, &block_3, &block_4};

  stk::mesh::Selector allNodes = stk::mesh::selectUnion(allBlocks);
  stk::mesh::Selector onlyCenterNode = stk::mesh::selectIntersection(allBlocks);
  stk::mesh::Selector nodes456 = (block_1 | block_2) & (block_3 | block_4);
  stk::mesh::Selector nodes123 = (block_1 | block_2) - nodes456;

  EXPECT_EQ(9u, stk::mesh::count_entities(*bulkPtr, stk::topology::NODE_RANK, allNodes));
  EXPECT_EQ(1u, stk::mesh::count_entities(*bulkPtr, stk::topology::NODE_RANK,
            onlyCenterNode));
  EXPECT_EQ(3u, stk::mesh::count_entities(*bulkPtr, stk::topology::NODE_RANK, nodes456));
  EXPECT_EQ(3u, stk::mesh::count_entities(*bulkPtr, stk::topology::NODE_RANK, nodes123));
}

TEST(StkMeshHowTo, betterUnderstandSelectorConstruction)
{
  MPI_Comm communicator = MPI_COMM_WORLD;
```

```
  if (stk::parallel_machine_size(communicator) != 1) { GTEST_SKIP(); }
  std::unique_ptr<stk::mesh::BulkData> bulkPtr = stk::mesh::MeshBuilder(communicator).create();
  const std::string generatedCubeMeshSpecification = "generated:1x1x1";
  stk::io::fill_mesh(generatedCubeMeshSpecification, *bulkPtr);

  stk::mesh::Selector nothingSelector_byDefaultConstruction;
  size_t expectingZeroBuckets = 0;
  EXPECT_EQ(expectingZeroBuckets, bulkPtr->get_buckets(stk::topology::NODE_RANK, nothingSelector_byDefaultConstruction).size());

  std::ostringstream readableSelectorDescription;
  readableSelectorDescription << nothingSelector_byDefaultConstruction;
  EXPECT_STREQ("NOTHING", readableSelectorDescription.str().c_str());

  stk::mesh::Selector allSelector(!nothingSelector_byDefaultConstruction);
  size_t numberOfAllNodeBuckets = bulkPtr->buckets(stk::topology::NODE_RANK).size();
  EXPECT_EQ(numberOfAllNodeBuckets, bulkPtr->get_buckets(stk::topology::NODE_RANK, allSelector).size());
}

TEST(StkMeshHowTo, makeSureYouAreNotIntersectingNothingSelector)
{
  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) != 1) { return; }
  std::unique_ptr<stk::mesh::BulkData> bulkPtr = stk::mesh::MeshBuilder(communicator).create();
  // syntax creates faces for surface on the positive: 'x-side', 'y-side', and 'z-side'
  // of a 1x1x1 cube, these parts are given the names: 'surface_1', 'surface_2', and 'surface_3'
  const std::string generatedCubeMeshSpecification = "generated:1x1x1|sideset:XYZ";
  stk::io::fill_mesh(generatedCubeMeshSpecification, *bulkPtr);

  stk::mesh::MetaData &stkMeshMetaData = bulkPtr->mesh_meta_data();
  stk::mesh::Part *surface1Part = stkMeshMetaData.get_part("surface_1");
  stk::mesh::Part *surface2Part = stkMeshMetaData.get_part("surface_2");
  stk::mesh::Part *surface3Part = stkMeshMetaData.get_part("surface_3");
  stk::mesh::PartVector allSurfaces;
  allSurfaces.push_back(surface1Part);
  allSurfaces.push_back(surface2Part);
  allSurfaces.push_back(surface3Part);

  stk::mesh::Selector selectorIntersectingNothing;
  for (size_t surfaceIndex = 0; surfaceIndex < allSurfaces.size(); ++surfaceIndex) {
    stk::mesh::Part &surfacePart = *(allSurfaces[surfaceIndex]);
    stk::mesh::Selector surfaceSelector(surfacePart);
    selectorIntersectingNothing &= surfacePart;
  }

  size_t expectedNumberOfBucketsWhenIntersectingNothing = 0;
  stk::mesh::BucketVector selectedBuckets = bulkPtr->get_buckets(stk::topology::NODE_RANK, selectorIntersectingNothing);
  EXPECT_EQ(expectedNumberOfBucketsWhenIntersectingNothing, selectedBuckets.size());

  stk::mesh::Selector preferredBoundaryNodesSelector = stk::mesh::selectIntersection(allSurfaces);
  size_t expectedNumberOfNodeBucketsWhenIntersectingAllSurfaces = 1;
  selectedBuckets = bulkPtr->get_buckets(stk::topology::NODE_RANK, preferredBoundaryNodesSelector);
  EXPECT_EQ(expectedNumberOfNodeBucketsWhenIntersectingAllSurfaces, selectedBuckets.size());
}
```

## 4.6. Mesh Modification

### *4.6.1. Overview*

The following types of mesh modifications are available in STK Mesh:

- Add/delete entities
- Change entities' part membership
- Change connectivity
- Change processors' entity ownership
- Change ghosting

A STK Mesh can be modified only within the context of a *modification cycle*. A modification cycle begins with a call to `BulkData::modification_begin()` and ends when the next call to `BulkData::modification_end()` returns. This latter function does a pre-determined set of checks on mesh status and performs MPI communication to ensure a globally-consistent state.

Modification cycles should not be nested; `BulkData::modification_end()` terminates all "enclosing" modification cycles. If the application inadvertently nests modification cycles, errors are likely to be thrown.

Application code between a `BulkData::modification_begin()` call and the following `BulkData::modification_end()` call can use STK Mesh modification functions that cause the BulkData to become parallel inconsistent. That is, mesh information on different processor ranks can disagree. After each modification cycle, a STK mesh is guaranteed to be parallel-consistent. Failures during mesh modification are not recoverable.

The first time `BulkData::modification_begin()` is called, the mesh MetaData is verified to have been committed and to be parallel-consistent (and the MetaData is committed at that time if it hasn't already been committed). The function returns **`true`** if the mesh successfully transitions from the guaranteed parallel-consistent state to the *MODIFIABLE* state, and **`false`** if it is already in this state.

`BulkData::modification_end()` performs parallel synchronization of local mesh modifications since the mesh entered the *MODIFIABLE* state and transitions the mesh back to a guaranteed parallel-consistent state. `BulkData::modification_end()` returns **`true`** if it succeeds and **`false`** if it is already in the guaranteed parallel-consistent state. If modification resolution errors occur then a parallel-consistent exception will be thrown.

Because a modification cycle incurs multiple rounds of communication and traversal over large portions of the mesh, even a modification cycle with a single modification incurs significant cost. From a performance standpoint it is advantageous to group mesh modifications into as few modification cycles as possible.

To alleviate the expense of a general modification cycle, other single-purpose API have been introduced, such as for the creation of faces, that take into account knowledge of what has been modified to improve the performance of a modification cycle. These should be considered before coding a general modification, especially if it is in a performance-critical part of the code.

Note that *MetaData* changes (declaring parts and fields) are not part of the mesh modification API since it's illegal to change MetaData after the MetaData object has been committed.

### 4.6.2. Public Modification Capability

In this section we describe the modification operations intended to be called from application code. As noted above, these functions can only be called between calls to `BulkData::modification_begin()` and `BulkData::modification_end()`. We also describe the modification operations that STK Mesh automatically performs internally as a result of an application explicitly calling a modification function. Understanding what modifications can occur automatically is particularly important for code reliability. We note that certain modification types are applicable only in distributed STK Mesh applications.

#### 4.6.2.1. Add/Delete Entities

The `BulkData::declare_entity()` function can be used to add an entity to a STK mesh and assign its entity rank and global identifier. `BulkData::generate_new_entities()` can be used to create multiple entities of specified entity ranks and have unique global identifiers automatically assigned. When entities of EDGE_RANK, FACE_RANK, or ELEMENT_RANK are created by application code, they must be assigned a topology and have their nodal connectivities set before `BulkData::modification_end()` is called. See section 4.6.6.

`BulkData::destroy_entity()` deletes an entity from a STK Mesh. All upward relations must be deleted before an entity can be destroyed, as a safety measure to ensure that the user is explicitly aware of any possible inconsistent mesh states that they are creating (e.g. an element that is missing one or more nodes). Downward relations are deleted automatically.

Adding or deleting an entity can result in automatic changes to part membership, ownership, connectivity, ghosting, and sharing. Changes in part membership(s) can also result in changes to bucket structure. Any local modifications to an entity will cause ghosted copies of that entity to be deleted from other processor ranks. The ghosts will be automatically regenerated if they are part of the aura.

Unless an entity is deleted, it stays valid before, during, and after a modification cycle.

**Listing 4.11 Example showing optimized destruction of all elements of a specified topology**
**code/stk/stk_doc_tests/stk_mesh/howToDestroyElementsOfTopology.cpp**

```
#include <gtest/gtest.h>
#include <stk_mesh/base/BulkData.hpp>
#include <stk_mesh/base/MeshBuilder.hpp>
#include <stk_mesh/base/MetaData.hpp>
#include <stk_mesh/base/GetEntities.hpp>
#include <stk_topology/topology.hpp>
#include <stk_io/FillMesh.hpp>
namespace
{
TEST(StkMeshHowTo, DestroyElementsOfTopology)
{
  if (stk::parallel_machine_size(MPI_COMM_WORLD) > 1) { GTEST_SKIP(); }

```

```
  std::unique_ptr<stk::mesh::BulkData> bulkPtr =
            stk::mesh::MeshBuilder(MPI_COMM_WORLD).create();
  stk::mesh::MetaData& metaData = bulkPtr->mesh_meta_data();
  stk::io::fill_mesh("generated:1x1x4", *bulkPtr);

  EXPECT_EQ(4u, stk::mesh::count_entities(*bulkPtr, stk::topology::ELEM_RANK,
            metaData.universal_part()));
  bulkPtr->destroy_elements_of_topology(stk::topology::HEX_8);
  EXPECT_EQ(0u, stk::mesh::count_entities(*bulkPtr, stk::topology::ELEM_RANK,
            metaData.universal_part()));
}
}
```

#### 4.6.2.2. Getting Unused Globally Unique Identifiers

Code Listing 4.12 shows, by example, how to get globally unique identifiers. The API requires that a stk topology rank be specified. The ids are then returned in the vector argument. These ids are unused when this call is made. Hence, care must be taken if these ids are kept on the application side (client side) and not used until later. This is a collective call (all processors must call this function). Note, this API is offered in addition to the `generate_new_entities()` method. The key difference is that the `generate_new_ids()` method only obtains identifiers per rank, and entities are not automatically created.

**Listing 4.12 Example showing how to use generate_new_ids**
**code/stk/stk_doc_tests/stk_mesh/howToUseGenerateNewIds.cpp**

```
TEST(StkMeshHowTo, use_generate_new_ids)
{
  MPI_Comm communicator = MPI_COMM_WORLD;

  int num_procs = stk::parallel_machine_size(communicator);
  std::unique_ptr<stk::mesh::BulkData> bulkPtr =
            stk::mesh::MeshBuilder(communicator).create();

  const std::string generatedMeshSpecification = "generated:1x1x" + std::to_string(num_procs);
  stk::io::fill_mesh(generatedMeshSpecification, *bulkPtr);

  // Given a mesh, request 10 unique node ids

  std::vector<stk::mesh::EntityId> requestedIds;
  unsigned numRequested = 10;

  bulkPtr->generate_new_ids(stk::topology::NODE_RANK, numRequested, requestedIds);

  test_that_ids_are_unique(*bulkPtr, stk::topology::NODE_RANK, requestedIds);
}
```

#### 4.6.2.3. Creating Nodes that are Shared by Multiple Processors

When a node entity is created that is intended to be shared by multiple processors (i.e., it will be connected to locally-owned entities on multiple MPI processors), the method `BulkData::add_node_sharing()` must be used to inform STK Mesh that the node is shared and which other processors share it. The `add_node_sharing()` method must be called symmetrically, meaning that for a given shared node, each sharing processor must inform

STK Mesh about all the other sharing processors during the same modification cycle. The code listing 4.13 demonstrates the use of `add_node_sharing()` when creating shared nodes.

**Listing 4.13 Example showing creation of shared nodes**
**code/stk/stk_doc_tests/stk_mesh/createSharedNodes.cpp**

```
TEST(stkMeshHowTo, createSharedNodes)
{
  const unsigned spatialDimension = 2;
  stk::mesh::MeshBuilder builder(MPI_COMM_WORLD);
  builder.set_spatial_dimension(spatialDimension);
  std::shared_ptr<stk::mesh::BulkData> bulkPtr = builder.create();
  stk::mesh::MetaData& metaData = bulkPtr->mesh_meta_data();
  stk::mesh::BulkData& bulkData = *bulkPtr;
  stk::mesh::Part &triPart = metaData.declare_part_with_topology("tri_part",
             stk::topology::TRIANGLE_3_2D);
  metaData.commit();

  if (bulkData.parallel_size() == 2)
  {
    bulkData.modification_begin();

    const unsigned nodesPerElem = 3;
    stk::mesh::EntityIdVector elemIds = {1, 2};//one elemId for each proc
    std::vector<stk::mesh::EntityIdVector> elemNodeIds = { {1, 3, 2}, {4, 2, 3} };
    const int myproc = bulkData.parallel_rank();

    stk::mesh::Entity elem = bulkData.declare_element(elemIds[myproc],
             stk::mesh::ConstPartVector{&triPart});
    stk::mesh::EntityVector elemNodes(nodesPerElem);
    elemNodes[0] = bulkData.declare_node(elemNodeIds[myproc][0]);
    elemNodes[1] = bulkData.declare_node(elemNodeIds[myproc][1]);
    elemNodes[2] = bulkData.declare_node(elemNodeIds[myproc][2]);

    bulkData.declare_relation(elem, elemNodes[0], 0);
    bulkData.declare_relation(elem, elemNodes[1], 1);
    bulkData.declare_relation(elem, elemNodes[2], 2);

    int otherproc = testUtils::get_other_proc(myproc);
    bulkData.add_node_sharing(elemNodes[1], otherproc);
    bulkData.add_node_sharing(elemNodes[2], otherproc);

    bulkData.modification_end();

    const size_t expectedTotalNumNodes = 4;
    verify_global_node_count(expectedTotalNumNodes, bulkData);
  }
}
```

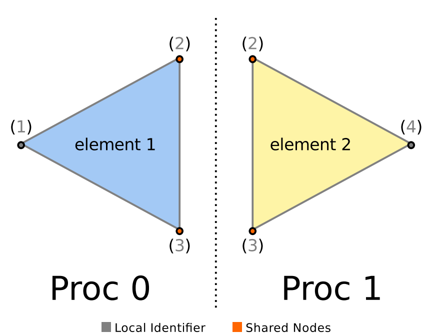



**Figure 4-7. Creation of shared nodes for code listing 4.13**

STK Mesh also supports the creation of independent shared nodes (nodes without connectivity) for use in p-refinement. In this case, additional nodes are created for higher order elements and these are maintained without explicit connectivity information in STK Mesh. Some of these nodes need to be shared across processor boundaries. This capability is to support the exploration of p-refinement. Currently, this capability cannot predict which nodes are attached to which elements when `change_entity_owner()` is called and therefore rebalance operations will likely not work as anticipated. This additional feature of `add_node_sharing()` is only enabled when the nodes are initially created. The code listing 4.14 demonstrates the use of `add_node_sharing()` to create independent shared nodes.

**Listing 4.14 Example showing creation of independent shared nodes**
**code/stk/stk_doc_tests/stk_mesh/createSharedNodes.cpp**

```cpp
TEST(stkMeshHowTo, createIndependentSharedNodes)
{
  const unsigned spatialDimension = 2;
  stk::mesh::MeshBuilder builder(MPI_COMM_WORLD);
  builder.set_spatial_dimension(spatialDimension);
  std::shared_ptr<stk::mesh::BulkData> bulkPtr = builder.create();
  stk::mesh::BulkData& bulkData = *bulkPtr;
  stk::mesh::MetaData& metaData = bulkPtr->mesh_meta_data();
  metaData.commit();

  if (bulkData.parallel_size() == 2)
  {
    bulkData.modification_begin();

    const unsigned nodesPerProc = 3;
    std::vector<stk::mesh::EntityIdVector> nodeIds = { {1, 3, 2}, {4, 2, 3} };
    const int myproc = bulkData.parallel_rank();
    stk::mesh::EntityVector nodes(nodesPerProc);
    nodes[0] = bulkData.declare_node(nodeIds[myproc][0]);
    nodes[1] = bulkData.declare_node(nodeIds[myproc][1]);
    nodes[2] = bulkData.declare_node(nodeIds[myproc][2]);

    int otherproc = testUtils::get_other_proc(myproc);
    bulkData.add_node_sharing(nodes[1], otherproc);
    bulkData.add_node_sharing(nodes[2], otherproc);

    bulkData.modification_end();

    const size_t expectedTotalNumNodes = 4;
    verify_global_node_count(expectedTotalNumNodes, bulkData);
  }
}
```

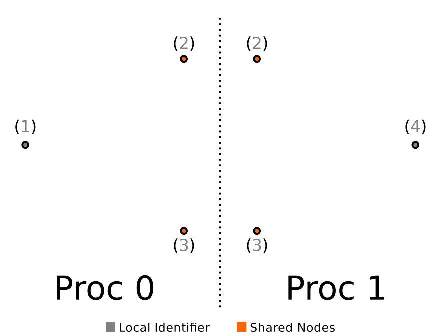



**Figure 4-8. creation of independent shared nodes for code listing 4.14**

This special marking to allow unconnected nodes to be shared will be removed if relations are attached to the node. The example 4.15 is a demonstration of this feature.

**Listing 4.15 Example showing independent shared nodes becoming dependent**
**code/stk/stk_doc_tests/stk_mesh/createSharedNodes.cpp**

```
TEST(stkMeshHowTo, createIndependentSharedNodesThenAddDependence)
{
  const unsigned spatialDimension = 2;
  stk::mesh::MeshBuilder builder(MPI_COMM_WORLD);
  builder.set_spatial_dimension(spatialDimension);
  std::shared_ptr<stk::mesh::BulkData> bulkPtr = builder.create();
  stk::mesh::BulkData& bulkData = *bulkPtr;
  stk::mesh::MetaData& metaData = bulkPtr->mesh_meta_data();
  stk::mesh::Part &triPart = metaData.declare_part_with_topology("triPart",
             stk::topology::TRIANGLE_3_2D);
  metaData.commit();

  if(bulkData.parallel_size() == 2)
  {
    bulkData.modification_begin();

    const unsigned nodesPerProc = 3;
    std::vector<stk::mesh::EntityIdVector> nodeIds = { {1, 3, 2}, {4, 2, 3}};
    const int myproc = bulkData.parallel_rank();

    stk::mesh::EntityVector nodes(nodesPerProc);
    nodes[0] = bulkData.declare_node(nodeIds[myproc][0]);
    nodes[1] = bulkData.declare_node(nodeIds[myproc][1]);
    nodes[2] = bulkData.declare_node(nodeIds[myproc][2]);

    int otherproc = testUtils::get_other_proc(myproc);
    bulkData.add_node_sharing(nodes[1], otherproc);
    bulkData.add_node_sharing(nodes[2], otherproc);

    const size_t expectedNumNodesPriorToModEnd = 6;
    verify_global_node_count(expectedNumNodesPriorToModEnd, bulkData);

    bulkData.modification_end();

    const size_t expectedNumNodesAfterModEnd = 4; // nodes 2 and 3 are shared
    verify_global_node_count(expectedNumNodesAfterModEnd, bulkData);

    const unsigned elemsPerProc = 1;
    stk::mesh::EntityId elemIds[][elemsPerProc] = { {1}, {2}};

    bulkData.modification_begin();
    stk::mesh::Entity elem = bulkData.declare_element(elemIds[myproc][0],
            stk::mesh::ConstPartVector{&triPart});
    bulkData.declare_relation(elem, nodes[0], 0);
    bulkData.declare_relation(elem, nodes[1], 1);
    bulkData.declare_relation(elem, nodes[2], 2);
    EXPECT_NO_THROW(bulkData.modification_end());

    bulkData.modification_begin();
    bulkData.destroy_entity(elem);
    bulkData.modification_end();

    if(myproc == 0)
      verify_nodes_2_and_3_are_no_longer_shared(bulkData, nodes);

    else  // myproc == 1
      verify_nodes_2_and_3_are_removed(bulkData, nodes);
  }
}
```

#### 4.6.2.4. Change Entity Part Membership

`BulkData::change_entity_parts()` changes which *parts* an entity belongs to.

Changes in part membership can result in changes to “induced” part membership. (See Section 4.4.2.) Changes in part membership typically cause entities to move to different buckets.

#### 4.6.2.5. Change Connectivity

`BulkData::declare_relation()` adds connectivity between two entities. `destroy_relation()` removes connectivity between two entities. Relations must be destroyed from the point of view of the higher-ranked entity toward the lower-ranked entity, although the relation in the other direction will also be removed automatically.

Changes in connectivity can result in changes to induced part membership. (See Section 4.4.2). Changes in connectivity can also result in changes in sharing and automatic ghosting during `modification_end()`. By causing changes in part membership(s), changes in connectivity can also result in changes to bucket structure.

#### 4.6.2.6. Change Entity Ownership

In a parallel mesh, it can be necessary to change what processor rank owns an entity. The typical case is when there is a change to parallel decomposition.

The `change_entity_owner` method is used for this and is called with a vector of pairs that specify entities and destination processors. It must be called on all processes even if the input vector is empty on some processors.

Changes in ownership can cause changes in ghosting and sharing, which are changes to part membership. By causing changes in part membership(s), changes in ownership can also result in changes to bucket structure.

#### 4.6.2.7. Change Ghosting

Aura ghosting is maintained automatically by STK Mesh, but can be optionally disabled. STK allows for application-specificed *custom ghosting*, through the functions `change_ghosting()`, `create_ghosting()`, `destroy_ghosting()`, and `destroy_all_ghosting()`. Each of these functions must be called parallel-synchronously.

The method `change_ghosting()` is used to add entities to be ghosted, or remove entities from a current ghosting. The input to the method includes a vector of pairs of entities and destination processors on which the entities are to be ghosted. To be added to a ghosting in this way, an entity must be locally-owned on the current processor, and must not already be shared by the destination processor. It is permissible for an entity to be in multiple different custom ghostings at the same time.

Any modification, directly applied or automatically called, to an entity in a ghosting will automatically cause that ghosting to be invalidated. For the aura ghosting, entities will be automatically regenerated during the next `modification_end()` call. For custom ghosting, it is not as well-defined what should happen to modified entities. It is possible for an entity in a ghosting to be invalidated without all of that ghosting being invalidated. `stk::mesh::BulkData::is_valid(entity)` can be used to determine whether a ghost entity has been invalidated.

### *4.6.3. Mesh Modification Examples*

Listing 4.16 shows how an element on processor 0 in the mesh depicted in Figure 4-9 is ghosted to processor 1. Note that Element 1 is connected to Node 1. This test shows how a user can use the identifier of the element, i.e. 1, to get an entity, and ghost it to another processor. This test also shows that Node 1 is automatically ghosted to processor 1 because it is a downward-relation of Element 1. In general, when an entity is ghosted, its downward-connected entities come along with it, but upward-connected entities don't.

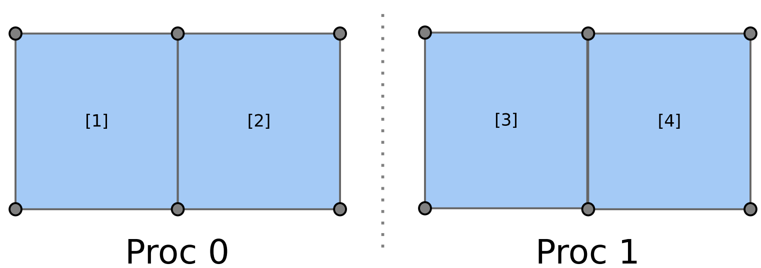



**Figure 4-9. Mesh Used in Listings 4.16-4.17**

**Listing 4.16 Example showing an element being ghosted**
**code/stk/stk_doc_tests/stk_mesh/customGhosting.cpp**

```
TEST(StkMeshHowTo, customGhostElem)
{
  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) == 2)
  {
    std::shared_ptr<stk::mesh::BulkData> bulkPtr =
            stk::mesh::MeshBuilder(communicator).create();
    stk::mesh::BulkData& bulkData = *bulkPtr;
    stk::io::fill_mesh("generated:1x1x4", bulkData);

    stk::mesh::EntityId id = 1;
    stk::mesh::Entity elem1 = bulkData.get_entity(stk::topology::ELEM_RANK, id);
    stk::mesh::Entity node1 = bulkData.get_entity(stk::topology::NODE_RANK, id);
    verify_that_elem1_and_node1_are_only_valid_on_p0(bulkData, elem1, node1);

    bulkData.modification_begin();
    stk::mesh::Ghosting& ghosting = bulkData.create_ghosting("custom ghost for elem 1");
    std::vector<std::pair<stk::mesh::Entity, int> > elemProcPairs;
    if (bulkData.parallel_rank() == 0)
      elemProcPairs.push_back(std::make_pair(elem1,
              get_other_proc(bulkData.parallel_rank())));
    bulkData.change_ghosting(ghosting, elemProcPairs);
    bulkData.modification_end();

```

```
    verify_that_elem1_and_downward_connected_entities_are_ghosted_from_p0_to_p1(bulkData, id);
  }
}

TEST(StkMeshHowTo, addElementToGhostingUsingSpecializedModificationForPerformance)
{
  MPI_Comm communicator = MPI_COMM_WORLD;
  if(stk::parallel_machine_size(communicator) == 2)
  {
    std::shared_ptr<stk::mesh::BulkData> bulkPtr =
            stk::mesh::MeshBuilder(communicator).create();
    stk::mesh::BulkData& bulk = *bulkPtr;
    stk::io::fill_mesh("generated:1x1x4", bulk);

    stk::mesh::EntityId elementId = 1;
    stk::mesh::Entity elem1 = bulk.get_entity(stk::topology::ELEM_RANK, elementId);
    verify_elem1_is_valid_only_on_p0(bulk, elem1);

    bulk.modification_begin();
    stk::mesh::Ghosting& ghosting = bulk.create_ghosting("my custom ghosting");
    bulk.modification_end();

    stk::mesh::EntityProcVec entityProcPairs;
    if(bulk.parallel_rank() == 0)
      entityProcPairs.push_back(stk::mesh::EntityProc(elem1,
            get_other_proc(bulk.parallel_rank())));

    bulk.batch_add_to_ghosting(ghosting, entityProcPairs);

    verify_elem1_is_valid_on_both_procs(bulk, elementId);
  }
}
```

Listing 4.17 shows how an entity can be moved, or stated alternatively, how to change an owner of an entity. Note that the `change_entity_owner()` method must be called by all processors, and must not be enclosed within calls to `modification_begin()` and `modification_end()` since it is a self-contained modification cycle.

**Listing 4.17 Example of changing processor ownership of an element**
**code/stk/stk_doc_tests/stk_mesh/changeEntityOwner.cpp**

```
TEST(StkMeshHowTo, changeEntityOwner)
{
  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) == 2)
  {
    std::shared_ptr<stk::mesh::BulkData> bulkDataPtr =
            stk::mesh::MeshBuilder(communicator).create();
    stk::io::fill_mesh("generated:1x1x4", *bulkDataPtr);

    stk::mesh::EntityId elem2Id = 2;
    stk::mesh::Entity elem2 = bulkDataPtr->get_entity(stk::topology::ELEM_RANK, elem2Id);
    verify_elem_is_owned_on_p0_and_valid_as_aura_on_p1(*bulkDataPtr, elem2);

    std::vector<std::pair<stk::mesh::Entity, int> > elemProcPairs;
    if (bulkDataPtr->parallel_rank() == 0)
      elemProcPairs.push_back(std::make_pair(elem2,
            testUtils::get_other_proc(bulkDataPtr->parallel_rank())));

    bulkDataPtr->change_entity_owner(elemProcPairs);

    verify_elem_is_now_owned_on_p1(*bulkDataPtr, elem2Id);
  }
}
```

#### 4.6.3.1. Resolving Sharing Of Exodus Sidesets - Special Case

Figure 4-10 shows a case of an interior Exodus sideset where two sides exist initially across a processor boundary. Nodes (1, 5, 8, 4) represent the face on the left (red) element on processor 0, and the nodes (1, 4, 8, 5) represent the face on the right (green) element on processor 1. The algorithm for determining if these two faces are the same shared face will consider the following two conditions:

1. The nodes on both face entities are the same or a valid permutation of each other
2. The identifiers of both face entities are the same

A boolean flag exists on BulkData, that if set to true, will require that two entities are the same if both conditions, (1) and (2), must be true for the entity to be marked as shared.

When reading an Exodus file and populating a STK Mesh, the current setting is that both conditions must be true for the mesh entities to be marked as the same. However, after the mesh has been read in, only condition (1) is used to resolve sharing of entities across parallel boundaries.

If the user desires one behavior over another, the `set_use_entity_ids_for_resolving_sharing()` function can be used before calling `modification_end()` during a mesh modification cycle. This behavior is undergoing changes so that the face entities created are consistently connected to elements. As such, the option discussed here is marked to be deprecated.

Code listing 4.18 shows two tests. The first test shows the option that can be used for resolving sharing. The second test case reads the mesh in Figure 4-10 and tests that there are two faces.

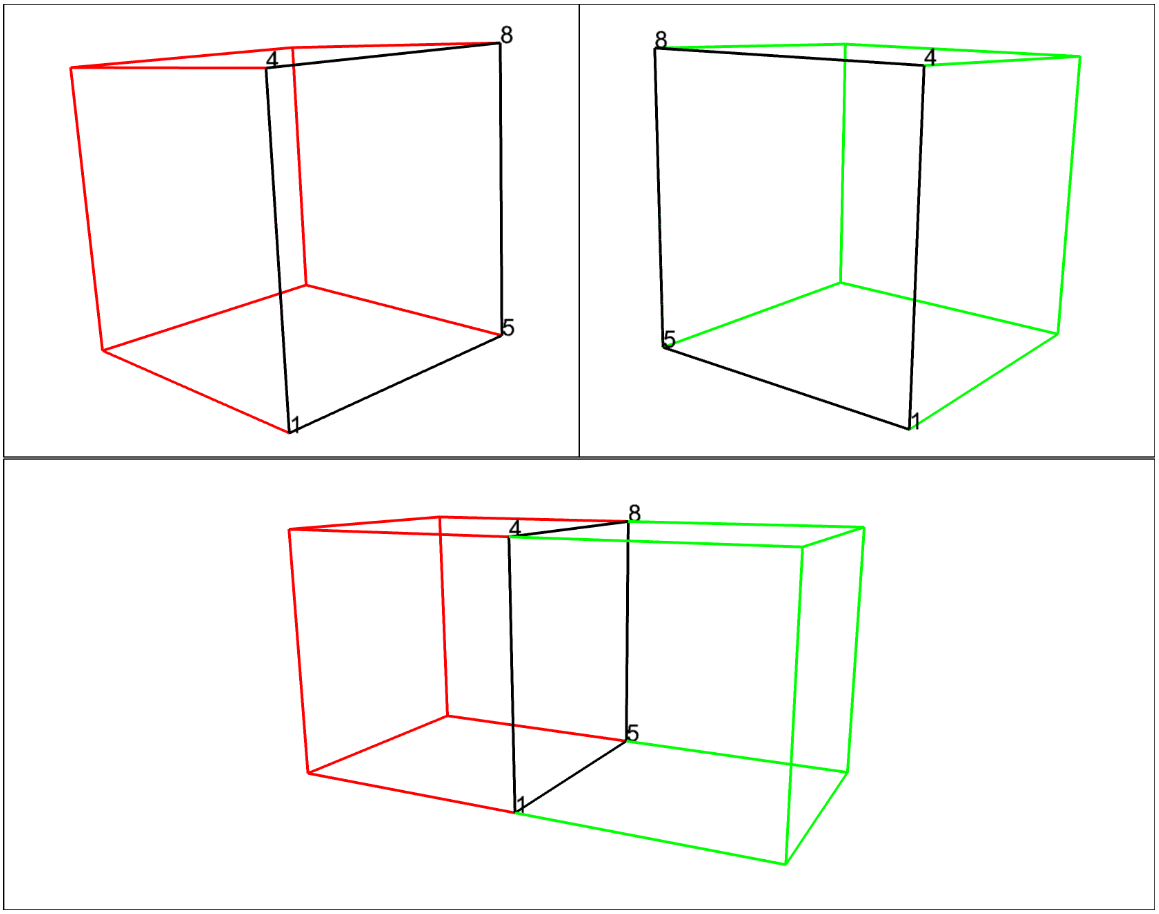



**Figure 4-10. Mesh Used in Listing 4.18**

**Listing 4.18 Example of internal sideset which results in two faces code/stk/stk_integration_tests/stk_mesh_doc/IntegrationTestBulkData.cpp**

```
TEST(BulkData_test, use_entity_ids_for_resolving_sharing)
{
    MPI_Comm communicator = MPI_COMM_WORLD;

    const int spatialDim = 3;
    stk::mesh::MetaData stkMeshMetaData(spatialDim);
    stk::unit_test_util::BulkDataTester stkMeshBulkData(stkMeshMetaData, communicator);

    if(stkMeshBulkData.parallel_size() == 2)
    {
        std::string exodusFileName = stk::unit_test_util::get_option("-i", "mesh.exo");

        stk::io::fill_mesh(exodusFileName, stkMeshBulkData);
    }

    stkMeshBulkData.set_use_entity_ids_for_resolving_sharing(false);
    EXPECT_FALSE(stkMeshBulkData.use_entity_ids_for_resolving_sharing());

    stkMeshBulkData.set_use_entity_ids_for_resolving_sharing(true);
    EXPECT_TRUE(stkMeshBulkData.use_entity_ids_for_resolving_sharing());
}

TEST(BulkData_test, testTwoDimProblemForSharingOfDifferentEdgesWithSameNodesFourProc)
{
```

```
    MPI_Comm communicator = MPI_COMM_WORLD;
    const int spatialDim = 2;
    stk::mesh::MetaData stkMeshMetaData(spatialDim);
    stk::unit_test_util::BulkDataTester stkMeshBulkData(stkMeshMetaData, communicator);

    if ( stkMeshBulkData.parallel_size() == 4 )
    {
        std::string exodusFileName = stk::unit_test_util::get_option("-i", "mesh.exo");

        stk::io::fill_mesh(exodusFileName, stkMeshBulkData);

        std::vector<size_t> globalCounts;
        stk::mesh::comm_mesh_counts(stkMeshBulkData, globalCounts);
        EXPECT_EQ(15u, globalCounts[stk::topology::EDGE_RANK]);
    }
}
```

### 4.6.4. Unsafe operations

There are a number of operations that are inherently unsafe to perform when the mesh is in the middle of a modification cycle. Exceptions will be thrown if the user tries to perform these operations during modification in a debug build, but not in a release build since the error checking is too expensive.

The *mesh_index* of an entity (which is a pairing of the entity's bucket and the entity's offset into that bucket) can be automatically changed by STK Mesh during a modification cycle. Thus, a mesh_index cannot be assumed to be valid during a modifcation cycle or be the same before and after it. A change in the membership of one or more buckets implies a change in the mesh index of one or more entities, and vice versa.

Although field data can be accessed during a modification cycle, parallel field operations (e.g., parallel sum) must be avoided during a modification cycle because the status of parallel sharing is not guaranteed to be globally consistent until after `BulkData::modification_end()`.

Mesh modification should generally not be done while looping over buckets. The problem is that mesh modification can cause entities to move from one bucket to another, which can invalidate the iteration over a particular bucket. Any loop that makes the assumption of Bucket stability, either the existence/order of a Bucket or the order of entities within the bucket, is not safe if the loop does mesh modification. Some errors that can result will be checked in debug, but never in release. If you must iterate the mesh and do mesh modification during the iteration, use an entity loop, not a bucket loop.

### 4.6.5. Automatic modification operations in modification_end()

When the client code is finished with all direct calls to any of the modifications in Section 4.6.2, it must call `modification_end()` to close the modification cycle.

`BulkData::modification_end()` automatically performs several types of modifications to the mesh to bring it into a parallel consistent state. These include

- Synchronizing entity membership in parts for shared entities.

- Refreshing the ghost layer around shared entities (referred to as the aura).
- Updating ghost entities in the aura that have changed part membership.
- Sorting buckets' entities for a well-defined ordering.
- Resolve side creation on the subdomain boundaries.

It is important to note that `modification_end()` used to automatically determine the sharing of nodes that had been created with the same global identifier on multiple MPI processors. It no longer does this, and client code is now required to inform STK Mesh of node sharing information. See section 4.6.2.3 for more details.

Since the sharing of entities is only changed automatically by STK Mesh internally, that functionality is not available through the STK Mesh API.

### 4.6.6. How to use generate_new_entities()

This example (Listing 4.19) shows how to use `BulkData::generate_new_entities()` to create new entities. After the entities are created, the ELEMENT_RANK entities are each assigned a topology and their nodal relations are set before `BulkData::modification_end()` is called. FACE_RANK and EDGE_RANK entities have the same requirement, but none are included in this example. The example also illustrates that it is incorrect to call `BulkData::modification_end()` if the requirement is not met.

**Listing 4.19 Example of how to generate multiple new entities and subsequently set topologies and nodal relations**
**code/stk/stk_doc_tests/stk_mesh/generateNewEntities.cpp**

```
TEST(stkMeshHowTo, generateNewEntities)
{
  const unsigned spatialDimension = 3;

  stk::mesh::MeshBuilder builder(MPI_COMM_WORLD);
  builder.set_spatial_dimension(spatialDimension);
  builder.set_entity_rank_names(stk::mesh::entity_rank_names());
  std::shared_ptr<stk::mesh::BulkData> bulkPtr = builder.create();
  stk::mesh::MetaData& metaData = bulkPtr->mesh_meta_data();
  stk::mesh::Part &tetPart = metaData.declare_part_with_topology("tetElementPart",
             stk::topology::TET_4);
  stk::mesh::Part &hexPart = metaData.declare_part_with_topology("hexElementPart",
             stk::topology::HEX_8);
  metaData.commit();

  // Parts vectors handy for setting topology later.
  std::vector<stk::mesh::Part *> add_tetPart(1);
  add_tetPart[0] = &tetPart;
  std::vector<stk::mesh::Part *> add_hexPart(1);
  add_hexPart[0] = &hexPart;

  stk::mesh::BulkData& mesh = *bulkPtr;
  mesh.modification_begin();

  std::vector<size_t> requests(metaData.entity_rank_count(), 0);
  const size_t num_nodes_requested = 12;
  const size_t num_elems_requested =  2;
  requests[stk::topology::NODE_RANK] = num_nodes_requested;
  requests[stk::topology::ELEMENT_RANK] = num_elems_requested;
  std::vector<stk::mesh::Entity> requested_entities;

```

```
  mesh.generate_new_entities(requests, requested_entities);

  // Set topologies of new entities with rank > stk::topology::NODE_RANK.
  stk::mesh::Entity elem1 = requested_entities[num_nodes_requested];
  mesh.change_entity_parts(elem1, add_tetPart);
  stk::mesh::Entity elem2 = requested_entities[num_nodes_requested + 1];
  mesh.change_entity_parts(elem2, add_hexPart);

  // Set downward relations of entities with rank > stk::topology::NODE_RANK
  unsigned node_i = 0;
  for(unsigned node_ord = 0 ; node_ord < 4; ++node_ord, ++node_i)
  {
    mesh.declare_relation( elem1 , requested_entities[node_i] , node_ord);
  }
  for(unsigned node_ord = 0 ; node_ord < 8; ++node_ord, ++node_i)
  {
    mesh.declare_relation( elem2 , requested_entities[node_i] , node_ord);
  }
  mesh.modification_end();

  check_connectivities_for_stkMeshHowTo_generateNewEntities(mesh, elem1, elem2,
              requested_entities);

  // Not setting topologies of new entities with rank > stk::topology::NODE_RANK causes throw
  mesh.modification_begin();
  std::vector<stk::mesh::Entity> more_requested_entities;
  mesh.generate_new_entities(requests, more_requested_entities);
#ifdef NDEBUG
  mesh.modification_end();
#else
  EXPECT_THROW(mesh.modification_end(), std::logic_error);
#endif
}
```

### 4.6.7. How to create faces

STK Mesh provides functions for creating all edges or faces for an existing mesh. This example demonstrates first creating a mesh of hex elements with nodes, (generated by STK IO), then uses the `create_faces()` function to create all faces in the mesh.

**Listing 4.20 Example of how to create all element faces**
**code/stk/stk_doc_tests/stk_mesh/createFacesHex.cpp**

```
TEST(StkMeshHowTo, CreateFacesHex)
{
  // ============================================================
  // INITIALIZATION
  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) != 1) { GTEST_SKIP(); }
  std::unique_ptr<stk::mesh::BulkData> bulkPtr =
              stk::mesh::MeshBuilder(communicator).create();

  const std::string generatedFileName = "generated:8x8x8";
  stk::io::fill_mesh(generatedFileName, *bulkPtr);

  // ============================================================
  //+ EXAMPLE
  //+ Create the faces..
  stk::mesh::create_faces(*bulkPtr);

  // ==================================================
  // VERIFICATION
```

```
  stk::mesh::Selector allEntities = bulkPtr->mesh_meta_data().universal_part();
  std::vector<size_t> entityCounts;
  stk::mesh::count_entities(allEntities, *bulkPtr, entityCounts);
  EXPECT_EQ( 512u, entityCounts[stk::topology::ELEMENT_RANK]);
  EXPECT_EQ(1728u, entityCounts[stk::topology::FACE_RANK]);

  // Edges are not generated, only faces.
  EXPECT_EQ(0u,   entityCounts[stk::topology::EDGE_RANK]);
}
```

### 4.6.8. How to create both edges and faces

This example demonstrates create all edges as well as faces for a hex-element mesh. Note that these functions only create relations to elements and nodes, so the faces will not have relations to the edges when both `create_edges()` and `create_faces()` are called.

**Listing 4.21 Example of how to create all element edges and faces**
**code/stk/stk_doc_tests/stk_mesh/createFacesEdgesHex.cpp**

```
TEST(StkMeshHowTo, CreateFacesEdgesHex)
{
  // ============================================================
  // INITIALIZATION
  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) != 1) { return; }
  stk::io::StkMeshIoBroker stkIo(communicator);

  const std::string generatedFileName = "generated:8x8x8";
  stkIo.add_mesh_database(generatedFileName, stk::io::READ_MESH);
  stkIo.create_input_mesh();
  stkIo.populate_bulk_data();

  // ============================================================
  //+ EXAMPLE
  //+ Create the faces..
  stk::mesh::create_faces(stkIo.bulk_data());

  //+ Create the edges..
  stk::mesh::create_edges(stkIo.bulk_data());

  // ==================================================
  // VERIFICATION
  stk::mesh::Selector allEntities = stkIo.meta_data().universal_part();
  std::vector<size_t> entityCounts;
  stk::mesh::count_entities(allEntities, stkIo.bulk_data(), entityCounts);
  EXPECT_EQ( 512u, entityCounts[stk::topology::ELEMENT_RANK]);
  EXPECT_EQ(1728u, entityCounts[stk::topology::FACE_RANK]);
  EXPECT_EQ(1944u, entityCounts[stk::topology::EDGE_RANK]);
  // MAKE SURE FACES ARE HOOKED TO EDGES
  // this should happen if create_faces is called before create_edges
  stk::mesh::BucketVector const & face_buckets =
             stkIo.bulk_data().buckets(stk::topology::FACE_RANK);
  for (size_t bucket_count=0, bucket_end=face_buckets.size(); bucket_count < bucket_end;
             ++bucket_count) {
    stk::mesh::Bucket & bucket = *face_buckets[bucket_count];
    const unsigned num_expected_edges = bucket.topology().num_edges();
    EXPECT_EQ(4u, num_expected_edges);
    for (size_t face_count=0, face_end=bucket.size(); face_count < face_end; ++face_count) {
      stk::mesh::Entity face = bucket[face_count];
      EXPECT_EQ(num_expected_edges, stkIo.bulk_data().num_edges(face));
    }
  }
```

```
}
```

### 4.6.9. How to create faces on only selected elements

This example demonstrates creating faces for a subset of the mesh elements defined by a Selector. Note that the "generated-mesh" syntax specifies that the initial mesh contains not only hex elements but also shell elements on all 6 sides.

**Listing 4.22 Example of how to create faces on only selected elements code/stk/stk_doc_tests/stk_mesh/createSelectedFaces.cpp**

```
TEST(StkMeshHowTo, CreateSelectedFacesHex)
{
  // ============================================================
  // INITIALIZATION
  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) != 1) { GTEST_SKIP(); }
  std::unique_ptr<stk::mesh::BulkData> bulkPtr =
            stk::mesh::MeshBuilder(communicator).create();

  // Generate a mesh containing 1 hex part and 6 shell parts
  const std::string generatedFileName = "generated:8x8x8|shell:xyzXYZ";
  stk::io::fill_mesh(generatedFileName, *bulkPtr);

  // ============================================================
  //+ EXAMPLE
  //+ Create a selector containing just the shell parts.
  stk::mesh::Selector shell_subset =
            bulkPtr->mesh_meta_data().get_topology_root_part(stk::topology::SHELL_QUAD_4);

  //+ Create the faces on just the selected shell parts.
  stk::mesh::create_all_sides(*bulkPtr, shell_subset);

  // ==================================================
  // VERIFICATION
  stk::mesh::Selector allEntities = bulkPtr->mesh_meta_data().universal_part();
  std::vector<size_t> entityCounts;
  stk::mesh::count_entities(allEntities, *bulkPtr, entityCounts);
  EXPECT_EQ( 896u, entityCounts[stk::topology::ELEMENT_RANK]);
  EXPECT_EQ( 768u, entityCounts[stk::topology::FACE_RANK]);

  // Edges are not generated, only faces.
  EXPECT_EQ(0u,   entityCounts[stk::topology::EDGE_RANK]);
}
```

### 4.6.10. Creating faces with layered shells

This example shows how many faces will be created when there are layered shells present.

**Listing 4.23 Example showing that faces are created correctly when layered shells are present code/stk/stk_doc_tests/stk_mesh/CreateFacesLayeredShellsHex.cpp**

```
TEST(StkMeshHowTo, CreateFacesLayeredShellsHex)
{
  // ============================================================
  // INITIALIZATION
  MPI_Comm communicator = MPI_COMM_WORLD;
```

```
  if (stk::parallel_machine_size(communicator) != 1) { return; }
  stk::io::StkMeshIoBroker stkIo(communicator);

  // Generate a mesh containing 1 hex part and 12 shell parts
  // Shells are layered 2 deep.
  const std::string generatedFileName = "generated:8x8x8|shell:xxyyzzXYZXYZ";
  stkIo.add_mesh_database(generatedFileName, stk::io::READ_MESH);
  stkIo.create_input_mesh();
  stkIo.populate_bulk_data();

  // ============================================================
  //+ EXAMPLE
  //+ Create the faces
  stk::mesh::create_faces(stkIo.bulk_data());

  // ==================================================
  // VERIFICATION
  stk::mesh::Selector allEntities = stkIo.meta_data().universal_part();
  std::vector<size_t> entityCounts;
  stk::mesh::count_entities(allEntities, stkIo.bulk_data(), entityCounts);
  EXPECT_EQ(1280u, entityCounts[stk::topology::ELEMENT_RANK]);
  //+ The shell faces are the same as the boundary hex faces
  EXPECT_EQ(2112u, entityCounts[stk::topology::FACE_RANK]);

  // Edges are not generated, only faces.
  EXPECT_EQ(0u,   entityCounts[stk::topology::EDGE_RANK]);
}
```

### 4.6.11. *Creating faces between hexes, on shells, and on shells between hexes*

This example shows how many faces are created on interior faces between hexes and shells.

**Listing 4.24 Example of how many faces get constructed by CreateFaces between two hexes code/stk/stk_doc_tests/stk_mesh/CreateFacesHexesShells.cpp**

```
TEST(StkMeshHowTo, CreateFacesTwoHexes)
{
  if (stk::parallel_machine_size(MPI_COMM_WORLD) == 1) {
    //   -----------
    //   |    |    |
    //   |HEX1|HEX2|
    //   |    |    |
    //   -----------
    stk::io::StkMeshIoBroker stkMeshIoBroker(MPI_COMM_WORLD);
    stkMeshIoBroker.add_mesh_database("AA.e", stk::io::READ_MESH);
    stkMeshIoBroker.create_input_mesh();
    stkMeshIoBroker.populate_bulk_data();
    stk::mesh::BulkData &mesh = stkMeshIoBroker.bulk_data();

    stk::mesh::create_all_sides(mesh, mesh.mesh_meta_data().universal_part());

    //   ------  F  ------
    //   |    |  A  |    |
    //   |HEX1|<-C->|HEX2|   Also external faces!
    //   |    |  E  |    |
    //   ------  !  ------

    unsigned first_bucket = 0;
    unsigned first_element_in_bucket = 0;
    stk::mesh::Entity first_element =
             (*mesh.buckets(stk::topology::ELEMENT_RANK)[first_bucket])[first_element_in_bucket];
    stk::mesh::Entity internal_face = mesh.begin_faces(first_element)[5];

```

```
    unsigned num_elements_connected_to_single_face = 2;
    EXPECT_EQ(num_elements_connected_to_single_face, mesh.num_elements(internal_face));

    unsigned num_expected_external_faces = 10u;
    unsigned num_expected_internal_faces = 1u;
    unsigned num_expected_faces = num_expected_external_faces + num_expected_internal_faces;
    stk::mesh::Selector all_entities = mesh.mesh_meta_data().universal_part();
    std::vector<size_t> entity_counts;
    stk::mesh::count_entities(all_entities, mesh, entity_counts);
    EXPECT_EQ(num_expected_faces, entity_counts[stk::topology::FACE_RANK]);
  }
}
```

**Listing 4.25 Example of how many faces get constructed by CreateFaces on a shell code/stk/stk_doc_tests/stk_mesh/CreateFacesHexesShells.cpp**

```
TEST(StkMeshHowTo, CreateFacesSingleShell)
{
  if (stk::parallel_machine_size(MPI_COMM_WORLD) == 1) {
    //  S
    //  H
    //  E
    //  L
    //  L
    stk::io::StkMeshIoBroker stkMeshIoBroker(MPI_COMM_WORLD);
    stkMeshIoBroker.add_mesh_database("e.e", stk::io::READ_MESH);
    stkMeshIoBroker.create_input_mesh();
    stkMeshIoBroker.populate_bulk_data();
    stk::mesh::BulkData &mesh = stkMeshIoBroker.bulk_data();

    stk::mesh::create_all_sides(mesh, mesh.mesh_meta_data().universal_part());

    //  F  S  F
    //  A  H  A
    //  C->E<-C
    //  E  L  E
    //  1  L  2

    unsigned first_bucket = 0;
    unsigned first_element_in_bucket = 0;
    stk::mesh::Entity first_element =
            (*mesh.buckets(stk::topology::ELEMENT_RANK)[first_bucket])[first_element_in_bucket];
    stk::mesh::Entity face_one = mesh.begin_faces(first_element)[0];
    unsigned num_elements_connected_to_face_one = 1;
    EXPECT_EQ(num_elements_connected_to_face_one, mesh.num_elements(face_one));

    stk::mesh::Entity face_two = mesh.begin_faces(first_element)[1];
    unsigned num_elements_connected_to_face_two = 1;
    EXPECT_EQ(num_elements_connected_to_face_two, mesh.num_elements(face_two));

    EXPECT_NE(face_one, face_two);

    unsigned num_expected_faces = 2u;
    stk::mesh::Selector all_entities = mesh.mesh_meta_data().universal_part();
    std::vector<size_t> entity_counts;
    stk::mesh::count_entities(all_entities, mesh, entity_counts);
    EXPECT_EQ(num_expected_faces, entity_counts[stk::topology::FACE_RANK]);
  }
}
```

**Listing 4.26 Example of how many faces get constructed by CreateFaces between hexes and an internal shell code/stk/stk_doc_tests/stk_mesh/CreateFacesHexesShells.cpp**

```
TEST(StkMeshHowTo, CreateFacesTwoHexesInternalShell)
```

```
{
  if (stk::parallel_machine_size(MPI_COMM_WORLD) == 1) {
    //  ------S------
    //  |    |H|    |
    //  |HEX1|E|HEX2|
    //  |    |L|    |
    //  ------L------
    stk::io::StkMeshIoBroker stkMeshIoBroker(MPI_COMM_WORLD);
    stkMeshIoBroker.add_mesh_database("AeA.e", stk::io::READ_MESH);
    stkMeshIoBroker.create_input_mesh();
    stkMeshIoBroker.populate_bulk_data();
    stk::mesh::BulkData &mesh = stkMeshIoBroker.bulk_data();

    stk::mesh::create_all_sides(mesh, mesh.mesh_meta_data().universal_part());

    //  ------  F  S  F  ------
    //  |    |  A  H  A  |    |
    //  |HEX1|<-C->E<-C->|HEX2|   Also external faces!
    //  |    |  E  L  E  |    |
    //  ------  1  L  2  ------

    unsigned first_bucket = 0;
    unsigned first_element_in_bucket = 0;
    stk::mesh::Entity first_element =
            (*mesh.buckets(stk::topology::ELEMENT_RANK)[first_bucket])[first_element_in_bucket];
    stk::mesh::Entity internal_face_one = mesh.begin_faces(first_element)[5];
    unsigned num_elements_connected_to_face_one = 2;
    EXPECT_EQ(num_elements_connected_to_face_one, mesh.num_elements(internal_face_one));

    unsigned second_element_in_bucket = 1;
    stk::mesh::Entity second_element =
            (*mesh.buckets(stk::topology::ELEMENT_RANK)[first_bucket])[second_element_in_bucket];
    stk::mesh::Entity internal_face_two = mesh.begin_faces(second_element)[4];
    unsigned num_elements_connected_to_face_two = 2;
    EXPECT_EQ(num_elements_connected_to_face_two, mesh.num_elements(internal_face_two));

    EXPECT_NE(internal_face_one, internal_face_two);

    unsigned num_expected_external_faces = 10u;
    unsigned num_expected_internal_faces = 2u;
    unsigned num_expected_faces = num_expected_external_faces + num_expected_internal_faces;
    stk::mesh::Selector all_entities = mesh.mesh_meta_data().universal_part();
    std::vector<size_t> entity_counts;
    stk::mesh::count_entities(all_entities, mesh, entity_counts);
    EXPECT_EQ(num_expected_faces, entity_counts[stk::topology::FACE_RANK]);
  }
}
```

### 4.6.12. How to skin a mesh

STK Mesh provides functions for skinning an existing mesh and creating appropriate boundary sides. This example demonstrates first creating a mesh of one hex element with nodes, (generated by STK IO), then uses the `create_exposed_boundary_sides()` function to skin the mesh.

**Listing 4.27 Example of how to create all the exposed boundary sides**
**code/stk/stk_doc_tests/stk_mesh/howToSkinMesh.cpp**

```
TEST(StkMeshHowTo, SkinExposedHex)
{
  // ============================================================
  // INITIALIZATION
```

```cpp
  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) != 1) { return; }

  std::shared_ptr<stk::mesh::BulkData> bulk = stk::mesh::MeshBuilder(communicator).create();
  stk::mesh::MetaData& meta = bulk->mesh_meta_data();

  const std::string generatedFileName = "generated:1x1x1";
  stk::io::fill_mesh(generatedFileName, *bulk);

  // ============================================================
  //+ EXAMPLE
  //+ Skin the mesh and create the exposed boundary sides..
  stk::mesh::Selector allEntities = meta.universal_part();
  stk::mesh::Part &skinPart = meta.declare_part("skin", meta.side_rank());
  stk::io::put_io_part_attribute(skinPart);

  stk::mesh::create_exposed_block_boundary_sides(*bulk, allEntities, {&skinPart});

  // ==================================================
  // VERIFICATION
  EXPECT_TRUE(stk::mesh::check_exposed_block_boundary_sides(*bulk, allEntities, skinPart));
  stk::mesh::Selector skin(skinPart & meta.locally_owned_part());
  unsigned numSkinnedSides = stk::mesh::count_entities(*bulk, meta.side_rank(), skin);
  EXPECT_EQ(6u, numSkinnedSides) << "in part " << skinPart.name();
}
```

### 4.6.13. How to create internal block boundaries of a mesh

STK Mesh also provides functions for creating the interior block boundary sides of an existing mesh. This example demonstrates first creating a mesh of two hex element with nodes, (generated by STK IO), creation of an IOPart into which element 2 is moved, followed by `create_interior_block_boundary_sides()` function to skin the mesh interior.

**Listing 4.28 Example of how to create all the interior block boundary sides code/stk/stk_doc_tests/stk_mesh/howToSkinMesh.cpp**

```cpp
TEST(StkMeshHowTo, SkinInteriorHex)
{
  // ============================================================
  // INITIALIZATION
  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) != 1) { return; }

  std::shared_ptr<stk::mesh::BulkData> bulk = stk::mesh::MeshBuilder(communicator).create();
  stk::mesh::MetaData& meta = bulk->mesh_meta_data();

  const std::string generatedFileName = "generated:1x1x2";
  stk::io::fill_mesh(generatedFileName, *bulk);

  // ============================================================
  //+ EXAMPLE
  //+ Skin the mesh and create the exposed boundary sides..
  stk::mesh::Selector allEntities = meta.universal_part();
  stk::mesh::Part &skinPart = meta.declare_part("skin", meta.side_rank());
  stk::io::put_io_part_attribute(skinPart);

  stk::mesh::Entity elem2 = bulk->get_entity(stk::topology::ELEM_RANK, 2u);
  stk::mesh::Part *block_1 = meta.get_part("block_1");

  bulk->modification_begin();
  stk::mesh::Part &block_2 = meta.declare_part("block_2", stk::topology::ELEM_RANK);
  stk::io::put_io_part_attribute(block_2);
```

```
  bulk->change_entity_parts(elem2, stk::mesh::ConstPartVector{&block_2},
             stk::mesh::ConstPartVector{block_1});
  bulk->modification_end();

  stk::mesh::create_interior_block_boundary_sides(*bulk, allEntities, {&skinPart});

  // ==================================================
  // VERIFICATION
  EXPECT_TRUE(stk::mesh::check_interior_block_boundary_sides(*bulk, allEntities, skinPart));
  stk::mesh::Selector skin(skinPart & meta.locally_owned_part());
  unsigned numSkinnedSides = stk::mesh::count_entities(*bulk, meta.side_rank(), skin);
  EXPECT_EQ(1u, numSkinnedSides) << "in part " << skinPart.name();
}
```

### 4.6.14. How to destroy elements in list

STK Mesh provides a means by which an application may destroy all the elements in a list as well as the downward connected entities in order to ensure that there are no orphaned nodes/faces.

**Listing 4.29 Example of how to destroy elements in a list**
**code/stk/stk_doc_tests/stk_mesh/howToDestroyElementsInList.cpp**

```
TEST(StkMeshHowTo, DestroyElementsInList)
{
  std::unique_ptr<stk::mesh::BulkData> bulkPtr =
             stk::mesh::MeshBuilder(MPI_COMM_WORLD).create();
  stk::mesh::BulkData& bulkData = *bulkPtr;
  stk::io::fill_mesh("generated:1x1x4", bulkData);
  EXPECT_GT(stk::mesh::count_entities(*bulkPtr, stk::topology::ELEM_RANK,
             bulkPtr->mesh_meta_data().universal_part()), 0u);
  stk::mesh::EntityVector
             elementsToDestroy{bulkData.get_entity(stk::topology::ELEMENT_RANK,1)};
  stk::mesh::destroy_elements(bulkData, elementsToDestroy);

  stk::mesh::EntityVector orphanedNodes{
    bulkData.get_entity(stk::topology::NODE_RANK,1),
    bulkData.get_entity(stk::topology::NODE_RANK,2),
    bulkData.get_entity(stk::topology::NODE_RANK,3),
    bulkData.get_entity(stk::topology::NODE_RANK,4)
  };

  for(stk::mesh::Entity node : orphanedNodes) {
    EXPECT_FALSE(bulkData.is_valid(node));
  }
}
```

## 4.7. STK Mesh usage examples

This section gives examples of how to access and manipulate a STK Mesh. The examples attempt to give demonstrations of several common tasks that an application developer may want to perform using STK Mesh.

### 4.7.1. How to iterate over nodes - Bucket loop vs `for_each_entity_run`

This pair of examples shows how to select the nodes for a subset of the mesh (a surface part), then iterate over those nodes and access the values of a temperature field associated with the nodes. The iteration is done two different ways. The first uses a bucket loop which takes advantage of the fact that field data is contiguous within a bucket. The second uses the `for_each_entity_run` mechanism which hides the mesh-traversal details and simply executes the user-supplied lambda or functor for each selected node. Note that the `for_each_entity_run` mechanism also uses a bucket loop internally, but it doesn't allow the efficiency gain of getting the field-data pointer once per bucket instead of once per entity. Usually that's a small gain but in some cases it can be significant.

**Listing 4.30 Two Examples of iterating over nodes**
**code/stk/stk_doc_tests/stk_mesh/howToIterateEntities.cpp**

```
TEST(StkMeshHowTo, iterateSidesetNodes_BucketLoop_ContiguousFieldDataWithinBucket)
{
  MPI_Comm comm = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(comm) != 1) { GTEST_SKIP(); }

  std::unique_ptr<stk::mesh::BulkData> stkMesh = stk::mesh::MeshBuilder(comm)
                                                   .set_spatial_dimension(3)
                                                   .create();
  stk::mesh::MetaData &stkMeshMeta = stkMesh->mesh_meta_data();
  stk::mesh::Field<double> &temperatureField =
            stkMeshMeta.declare_field<double>(stk::topology::NODE_RANK, "temperature");
  stk::mesh::put_field_on_entire_mesh(temperatureField);

  // syntax creates faces for the surface on the positive 'x-side' of the 2x2x2 cube,
  // this part is given the name 'surface_1' when it is created.
  const std::string generatedMeshSpecification = "generated:2x2x2|sideset:X";
  stk::io::fill_mesh(generatedMeshSpecification, *stkMesh);

  stk::mesh::Part &boundaryConditionPart = *stkMeshMeta.get_part("surface_1");
  stk::mesh::Selector boundaryNodesSelector(boundaryConditionPart);

  const stk::mesh::BucketVector &boundaryNodeBuckets =
            stkMesh->get_buckets(stk::topology::NODE_RANK, boundaryNodesSelector);

  constexpr double prescribedTemperatureValue = 2.0;
  for (size_t bucketIndex = 0; bucketIndex < boundaryNodeBuckets.size(); ++bucketIndex) {
    const stk::mesh::Bucket &nodeBucket = *boundaryNodeBuckets[bucketIndex];
    double *temperatureValues = stk::mesh::field_data(temperatureField, nodeBucket);

    for (size_t nodeIndex = 0; nodeIndex < nodeBucket.size(); ++nodeIndex) {
      temperatureValues[nodeIndex] = prescribedTemperatureValue;
    }
  }

  testUtils::testTemperatureFieldSetCorrectly(temperatureField, boundaryNodesSelector,
            prescribedTemperatureValue);
}

TEST(StkMeshHowTo, iterateSidesetNodes_ForEachEntity_FieldDataAccess)
{
  MPI_Comm comm = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(comm) != 1) { GTEST_SKIP(); }

  std::unique_ptr<stk::mesh::BulkData> stkMesh = stk::mesh::MeshBuilder(comm)
                                                   .set_spatial_dimension(3)
                                                   .create();
  stk::mesh::MetaData &stkMeshMeta = stkMesh->mesh_meta_data();
```

```
  stk::mesh::Field<double> &temperatureField =
             stkMeshMeta.declare_field<double>(stk::topology::NODE_RANK, "temperature");
  stk::mesh::put_field_on_entire_mesh(temperatureField);

  // syntax creates faces for the surface on the positive 'x-side' of the 2x2x2 cube,
  // this part is given the name 'surface_1' when it is created.
  const std::string generatedMeshSpecification = "generated:2x2x2|sideset:X";
  stk::io::fill_mesh(generatedMeshSpecification, *stkMesh);

  stk::mesh::Part &boundaryConditionPart = *stkMeshMeta.get_part("surface_1");
  stk::mesh::Selector boundaryNodesSelector(boundaryConditionPart);

  constexpr double prescribedTemperatureValue = 2.0;

  stk::mesh::for_each_entity_run(*stkMesh, stk::topology::NODE_RANK, boundaryNodesSelector,
    [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
      double *temperatureValues = stk::mesh::field_data(temperatureField, node);
      *temperatureValues = prescribedTemperatureValue;
    });

  testUtils::testTemperatureFieldSetCorrectly(temperatureField, boundaryNodesSelector,
             prescribedTemperatureValue);
}
```

#### 4.7.2. *How to traverse mesh connectivity*

The following three examples show different ways to traverse elements and element-to-node connectivity.

The first is a simple usage of `for_each_entity_run_with_nodes` that addresses the common case of nodal connectivity specifically.

The second example uses `for_each_entity_run` and obtains nodal connectivity from `BulkData` in a more general way that could easily be adapted to other kinds of traversals such as element-to-face, node-to-element, etc.

Finally, the third example demonstrates the traversal using a bucket loop and accessing connectivity through `Bucket` APIs. In certain scenarios, this could be the most efficient method, since `BulkData` methods must first look up the bucket for the given entity and rank and the entity's index in that bucket.

**Listing 4.31 Examples of how to traverse connectivity via accessors on BulkData and via accessors on Bucket**
**code/stk/stk_doc_tests/stk_mesh/howToIterateConnectivity.cpp**

```
TEST(StkMeshHowTo, iterateElemNodeConnectivity_ForEachEntityWithNodes)
{
  MPI_Comm comm = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(comm) != 1) { GTEST_SKIP(); }
  std::unique_ptr<stk::mesh::BulkData> stkMesh = stk::mesh::MeshBuilder(comm).create();
  // Generate a mesh of unit-cube hexes with a sideset
  const std::string generatedMeshSpecification = "generated:2x2x2|sideset:X";
  stk::io::fill_mesh(generatedMeshSpecification, *stkMesh);

  typedef stk::mesh::Field<double> CoordinatesField_t;
  CoordinatesField_t const & coord_field =
      *dynamic_cast<CoordinatesField_t const *>(stkMesh->mesh_meta_data().coordinate_field());

  constexpr unsigned nodesPerHex = 8;
```

```
  constexpr unsigned spatialDim = 3;
  unsigned count = 0;
  double elementNodeCoords[nodesPerHex][spatialDim] = {
    {NAN,NAN,NAN}, {NAN,NAN,NAN}, {NAN,NAN,NAN}, {NAN,NAN,NAN},
    {NAN,NAN,NAN}, {NAN,NAN,NAN}, {NAN,NAN,NAN}, {NAN,NAN,NAN} };

  stk::mesh::Selector all = stkMesh->mesh_meta_data().universal_part();

  stk::mesh::for_each_entity_run_with_nodes(*stkMesh, stk::topology::ELEM_RANK, all,
    [&](stk::mesh::Entity elem, const stk::mesh::Entity* nodes, size_t numNodesPerEntity) {
      EXPECT_EQ(numNodesPerEntity, nodesPerHex);
      for (unsigned inode = 0; inode < numNodesPerEntity; ++inode) {
        const double *coords = stk::mesh::field_data(coord_field, nodes[inode]);
        elementNodeCoords[inode][0] = coords[0];
        elementNodeCoords[inode][1] = coords[1];
        elementNodeCoords[inode][2] = coords[2];
        ++count;
      }
    });

  const unsigned numElems = 2*2*2;
  const unsigned totalNodesVisited = numElems * nodesPerHex;
  EXPECT_EQ(count, totalNodesVisited);
  EXPECT_FALSE(std::isnan(elementNodeCoords[0][0]));
}

TEST(StkMeshHowTo, iterateConnectivity_General_BulkData)
{
  MPI_Comm comm = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(comm) != 1) { GTEST_SKIP(); }
  std::unique_ptr<stk::mesh::BulkData> stkMesh = stk::mesh::MeshBuilder(comm).create();
  // Generate a mesh of unit-cube hexes with a sideset
  const std::string generatedMeshSpecification = "generated:2x2x2|sideset:X";
  stk::io::fill_mesh(generatedMeshSpecification, *stkMesh);

  typedef stk::mesh::Field<double> CoordinatesField_t;
  CoordinatesField_t const & coord_field =
      *dynamic_cast<CoordinatesField_t const *>(stkMesh->mesh_meta_data().coordinate_field());

  constexpr unsigned nodesPerHex = 8;
  constexpr unsigned spatialDim = 3;
  unsigned count = 0;
  double elementNodeCoords[nodesPerHex][spatialDim] = {
    {NAN,NAN,NAN}, {NAN,NAN,NAN}, {NAN,NAN,NAN}, {NAN,NAN,NAN},
    {NAN,NAN,NAN}, {NAN,NAN,NAN}, {NAN,NAN,NAN}, {NAN,NAN,NAN} };

  stk::mesh::for_each_entity_run(*stkMesh, stk::topology::ELEM_RANK,
    [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity elem) {
      const stk::mesh::ConnectedEntities nodes = stkMesh->get_connected_entities(elem,
            stk::topology::NODE_RANK);
      EXPECT_EQ(nodes.size(), nodesPerHex);

      for (unsigned inode = 0; inode < nodes.size(); ++inode) {
        const double *coords = stk::mesh::field_data(coord_field, nodes[inode]);
        elementNodeCoords[inode][0] = coords[0];
        elementNodeCoords[inode][1] = coords[1];
        elementNodeCoords[inode][2] = coords[2];
        ++count;
      }
    });

  const unsigned numElems = 2*2*2;
  const unsigned totalNodesVisited = numElems * nodesPerHex;
  EXPECT_EQ(count, totalNodesVisited);
  EXPECT_FALSE(std::isnan(elementNodeCoords[0][0]));
}

TEST(StkMeshHowTo, iterateConnectivity_Buckets)
```

```
{
  MPI_Comm comm = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(comm) != 1) { return; }
  std::unique_ptr<stk::mesh::BulkData> stkMesh = stk::mesh::MeshBuilder(comm).create();
  // Generate a mesh of unit-cube hexes with a sideset
  const std::string generatedMeshSpecification = "generated:2x2x2|sideset:X";
  stk::io::fill_mesh(generatedMeshSpecification, *stkMesh);

  typedef stk::mesh::Field<double> CoordinatesField_t;
  CoordinatesField_t const & coord_field =
      *dynamic_cast<CoordinatesField_t const *>(stkMesh->mesh_meta_data().coordinate_field());

  const stk::mesh::BucketVector &elementBuckets =
            stkMesh->buckets(stk::topology::ELEMENT_RANK);

  constexpr unsigned nodesPerHex = 8;
  constexpr unsigned spatialDim = 3;
  unsigned count = 0;
  double elementNodeCoords[nodesPerHex][spatialDim] = {
    {NAN,NAN,NAN}, {NAN,NAN,NAN}, {NAN,NAN,NAN}, {NAN,NAN,NAN},
    {NAN,NAN,NAN}, {NAN,NAN,NAN}, {NAN,NAN,NAN}, {NAN,NAN,NAN} };

  for (size_t bucketIndex = 0; bucketIndex < elementBuckets.size(); ++bucketIndex) {
    const stk::mesh::Bucket &elemBucket = *elementBuckets[bucketIndex];

    for (size_t elemIndex = 0; elemIndex < elemBucket.size(); ++elemIndex) {
      unsigned numNodes = elemBucket.num_nodes(elemIndex);
      EXPECT_EQ(numNodes, nodesPerHex);
      const stk::mesh::Entity* nodes = elemBucket.begin_nodes(elemIndex);

      for (unsigned inode = 0; inode < numNodes; ++inode) {
        const double *coords = stk::mesh::field_data(coord_field, nodes[inode]);
        elementNodeCoords[inode][0] = coords[0];
        elementNodeCoords[inode][1] = coords[1];
        elementNodeCoords[inode][2] = coords[2];
        ++count;
      }
    }
  }
  const unsigned numElems = 2*2*2;
  const unsigned totalNodesVisited = numElems * nodesPerHex;
  EXPECT_EQ(count, totalNodesVisited);
  EXPECT_FALSE(std::isnan(elementNodeCoords[0][0]));
}
```

### 4.7.3. How to check side equivalency

**Listing 4.32 Example of how to check side equivalency**
**code/stk/stk_doc_tests/stk_mesh/howToUseEquivalent.cpp**

```
TEST_F(MeshWithSide, whenCheckingSideEquivalency_returnsCorrectPermutation)
{
  if (stk::parallel_machine_size(get_comm()) == 1) {
    setup_mesh("generated:1x1x4|sideset:x", stk::mesh::BulkData::NO_AUTO_AURA);
    stk::mesh::Entity elem1 = get_bulk().get_entity(stk::topology::ELEM_RANK, 1);
    ASSERT_EQ(1u, get_bulk().num_faces(elem1));
    const stk::mesh::Entity side = *get_bulk().begin_faces(elem1);
    const stk::mesh::Permutation perm = *get_bulk().begin_face_permutations(elem1);
    const stk::mesh::ConnectivityOrdinal ordinal = *get_bulk().begin_face_ordinals(elem1);
    const stk::mesh::Entity* sideNodes = get_bulk().begin_nodes(side);
    unsigned numNodes = get_bulk().num_nodes(side);

    stk::EquivalentPermutation equivAndPermutation = stk::mesh::side_equivalent(get_bulk(),
            elem1, ordinal, sideNodes);
```

```
    EXPECT_TRUE(equivAndPermutation.is_equivalent);
    EXPECT_EQ(perm,
            static_cast<stk::mesh::Permutation>(equivAndPermutation.permutation_number));

    EXPECT_TRUE(stk::mesh::is_side_equivalent(get_bulk(), elem1, ordinal, sideNodes));

    stk::mesh::EquivAndPositive result =
            stk::mesh::is_side_equivalent_and_positive(get_bulk(), elem1, ordinal,
            sideNodes, numNodes);
    EXPECT_TRUE(result.is_equiv);
    EXPECT_TRUE(result.is_positive);
  }
}
```

### 4.7.4. Understanding node ordering of edges and faces

Listing 4.33 shows the difference between node orderings when using the STK Mesh `create_edges()` and `create_faces()` functions versus STK Topology. Listing 3.10 has more information regarding the lexicographical smallest permutation which is used to change the ordering for the two cases.

**Listing 4.33 Understanding edge and face ordering**
**code/stk/stk_doc_tests/stk_mesh/createFacesEdgesHex.cpp**

```
// ============================================================
//+ EXAMPLE
//+ Create the faces..
stk::mesh::create_faces(bulkData);

unsigned goldValuesForHexFaceNodesFromStkTopology[6][4] = {
  {1, 2, 6, 5}, {2, 3, 7, 6}, {3, 4, 8, 7}, {1, 5, 8, 4}, {1, 4, 3, 2}, {5, 6, 7, 8} };

// Lexicographical smallest permutation per face leads from topology ordering (above) for
          face to ordering below

unsigned goldValuesForHexFaceNodesFromCreateFaces[6][4] = {
  {1, 2, 6, 5}, {2, 3, 7, 6}, {3, 4, 8, 7}, {1, 4, 8, 5}, {1, 2, 3, 4}, {5, 6, 7, 8} };

//+ Create the edges..
stk::mesh::create_edges(bulkData);

unsigned goldValuesHexEdgeNodesFromStkTopology[12][2] = {
  {1, 2}, {2, 3}, {3, 4}, {4, 1}, {5, 6}, {6, 7}, {7, 8}, {8, 5}, {1, 5}, {2, 6}, {3, 7},
          {4, 8} };

// Lexicographical smallest permutation per edge leads from topology ordering (above) for
          edge to ordering below

unsigned goldValuesHexEdgeNodesFromCreateEdges[12][2] = {
  {1, 2}, {2, 3}, {3, 4}, {1, 4}, {5, 6}, {6, 7}, {7, 8}, {5, 8}, {1, 5}, {2, 6}, {3, 7},
          {4, 8} };


```

### 4.7.5. How to sort entities into an arbitrary order

One possible use case for this is to try and improve cache hit rate when visiting the nodes of an element.

**Listing 4.34 Example showing how to sort entities by descending identifier**
**code/stk/stk_doc_tests/stk_mesh/howToSortEntities.cpp**

```
#include "gtest/gtest.h"
#include <stk_mesh/base/BulkData.hpp>
#include <stk_mesh/base/EntitySorterBase.hpp>
#include <stk_unit_test_utils/MeshFixture.hpp>

namespace {

class EntityReverseSorter : public stk::mesh::EntitySorterBase
{
public:
  virtual void sort(stk::mesh::BulkData &bulk, stk::mesh::EntityVector& entityVector) const
  {
    std::sort(entityVector.begin(), entityVector.end(),
              [&bulk](stk::mesh::Entity a, stk::mesh::Entity b) { return bulk.identifier(a) >
             bulk.identifier(b); });
  }
};

class HowToSortEntities : public stk::unit_test_util::MeshFixture
{
protected:
  void sort_and_check()
  {
    if(stk::parallel_machine_size(get_comm()) == 1)
    {
      setup_mesh("generated:1x1x4", stk::mesh::BulkData::AUTO_AURA);
      get_bulk().sort_entities(EntityReverseSorter());
      expect_entities_in_reverse_order();
    }
  }
  void expect_entities_in_reverse_order()
  {
    const stk::mesh::BucketVector buckets = get_bulk().buckets(stk::topology::NODE_RANK);
    ASSERT_EQ(1u, buckets.size());
    expect_bucket_in_reverse_order(*buckets[0]);
  }
  void expect_bucket_in_reverse_order(const stk::mesh::Bucket &bucket)
  {
    ASSERT_EQ(20u, bucket.size());
    for(size_t i=1; i<bucket.size(); i++)
      EXPECT_GT(get_bulk().identifier(bucket[i-1]), get_bulk().identifier(bucket[i]));
  }
};
TEST_F(HowToSortEntities, example_reverse)
{
  sort_and_check();
}

}
```

## 4.8. STK Fields

A STK *field* is a data structure that defines values associated with entities, such as temperatures, coordinates, or stress. A field can be defined over the whole mesh or a subset of the mesh (typically defined by a list of parts). STK Mesh currently manages STK field creation, storage, retrieval and field data memory allocation. Fields are managed by entity rank (node, edge, face, element, etc.), meaning that a given Field is only allocated on a single entity rank. Multiple Fields can have the same name as long as they are defined on different entity ranks.

The following code listings demonstrate some common usage of fields:

- Scalar, vector, and tensor fields
- Fields on nodes or on elements
- Fields allocated for the entire mesh
- Fields allocated for only part of the mesh
- Fields with constant size across the mesh
- Fields with variable size per part
- Multi-State fields
- Communicate field data

In each example, the general flow of execution is as follows:

1. Declare and initialize `stk::mesh::MetaData`: declare fields and parts
2. Declare and initialize `stk::mesh::BulkData`: create elements and nodes
3. Initialize, access and/or test field-data.

## 4.9. Example STK fields usage

**Listing 4.35 Examples of constant-size whole-mesh field usage**
**code/stk/stk_doc_tests/stk_mesh/useSimpleFields.cpp**

```
TEST(stkMeshHowTo, useSimpleFields)
{
  if (stk::parallel_machine_size(MPI_COMM_WORLD) > 1) { GTEST_SKIP(); }

  stk::mesh::MeshBuilder builder(MPI_COMM_WORLD);
  builder.set_spatial_dimension(SpatialDimension);
  std::unique_ptr<stk::mesh::BulkData> bulkPtr = builder.create();
  stk::mesh::MetaData& metaData = bulkPtr->mesh_meta_data();

  typedef stk::mesh::Field<double> DoubleField;
  DoubleField& pressureField = metaData.declare_field<double>(stk::topology::ELEM_RANK,
           "pressure");
  DoubleField& displacementsField = metaData.declare_field<double>(stk::topology::NODE_RANK,
           "displacements");

  constexpr double initialPressureValue = 4.4;
  constexpr unsigned vectorFieldLengthPerEntity = 3;
  stk::mesh::put_field_on_entire_mesh_with_initial_value(pressureField,
           &initialPressureValue);
  stk::mesh::put_field_on_mesh(displacementsField, metaData.universal_part(),
           vectorFieldLengthPerEntity, nullptr);
  stk::io::set_field_output_type(displacementsField, stk::io::FieldOutputType::VECTOR_3D);

  stk::mesh::BulkData& mesh = *bulkPtr;
  create_two_tet_element_mesh(mesh);

  auto expectEqualZero = [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
    const double* displacementDataForNode = stk::mesh::field_data(displacementsField, node);
    for(unsigned i=0; i<vectorFieldLengthPerEntity; ++i) {
      EXPECT_EQ(0.0, displacementDataForNode[i]);
    }
  };

```

```
  stk::mesh::for_each_entity_run(mesh, stk::topology::NODE_RANK, expectEqualZero);

  stk::mesh::field_fill(99.0, displacementsField);

  auto expectEqual99 = [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
    const double* displacementDataForNode = stk::mesh::field_data(displacementsField, node);
    for(unsigned i=0; i<vectorFieldLengthPerEntity; ++i) {
      EXPECT_EQ(99.0, displacementDataForNode[i]);
    }
  };

  stk::mesh::for_each_entity_run(mesh, stk::topology::NODE_RANK, expectEqual99);

  stk::mesh::Entity elem1 = mesh.get_entity(stk::topology::ELEM_RANK, 1);
  double* pressureFieldDataForElem1 = stk::mesh::field_data(pressureField, elem1);
  EXPECT_EQ(initialPressureValue, *pressureFieldDataForElem1);

  stk::mesh::Entity elem2 = mesh.get_entity(stk::topology::ELEM_RANK, 2);
  double* pressureFieldDataForElem2 = stk::mesh::field_data(pressureField, elem2);
  EXPECT_EQ(initialPressureValue, *pressureFieldDataForElem2);
}
```

**Listing 4.36 Examples of how to get fields by name**
**code/stk/stk_doc_tests/stk_mesh/howToGetFields.cpp**

```
TEST(stkMeshHowTo, getFields)
{
  stk::mesh::MetaData metaData(SpatialDimension::three);

  typedef stk::mesh::Field<double> DoubleFieldType;

  const std::string pressureFieldName = "pressure";
  DoubleFieldType *pressureField = &metaData.declare_field<double>(stk::topology::ELEM_RANK,
           pressureFieldName);
  metaData.commit();

  EXPECT_EQ(pressureField, metaData.get_field<double>(stk::topology::ELEM_RANK,
           pressureFieldName));
  EXPECT_EQ(pressureField, metaData.get_field(stk::topology::ELEM_RANK, pressureFieldName));
}
```

**Listing 4.37 Examples of using fields that are variable-size and defined on only a subset of the mesh**
**code/stk/stk_doc_tests/stk_mesh/useAdvancedFields.cpp**

```
TEST(stkMeshHowTo, useAdvancedFields)
{
  if (stk::parallel_machine_size(MPI_COMM_WORLD) > 1) { GTEST_SKIP(); }

  const unsigned spatialDimension = 3;
  stk::mesh::MeshBuilder builder(MPI_COMM_WORLD);
  builder.set_spatial_dimension(spatialDimension);
  std::unique_ptr<stk::mesh::BulkData> bulkPtr = builder.create();
  stk::mesh::MetaData& metaData = bulkPtr->mesh_meta_data();

  typedef stk::mesh::Field<double> DoubleField;
  DoubleField& tensorField = metaData.declare_field<double>(stk::topology::ELEM_RANK,
           "tensor");
  DoubleField& variableSizeField = metaData.declare_field<double>(stk::topology::ELEM_RANK,
           "variableSizeField");

  stk::mesh::Part &tetPart = metaData.declare_part_with_topology("tetElementPart",
           stk::topology::TET_4);
  stk::mesh::Part &hexPart = metaData.declare_part_with_topology("hexElementPart",
           stk::topology::HEX_8);
```

```
const int numTensorValues = 9;
const int numCopies = 2;
double initialTensorValue[] = { 1,  2,  3,  4,  5,  6,  7,  8,  9,
                               11, 12, 13, 14, 15, 16, 17, 18, 19};
stk::mesh::put_field_on_mesh(tensorField, metaData.universal_part(), numTensorValues,
          numCopies, initialTensorValue);
stk::io::set_field_output_type(tensorField, stk::io::FieldOutputType::FULL_TENSOR_36);

const int numVectorValues = 3;
double initialVectorValue[] = {1, 2, 3, 11, 12, 13};
stk::mesh::put_field_on_mesh(variableSizeField, tetPart, numVectorValues,
          initialVectorValue);
stk::mesh::put_field_on_mesh(variableSizeField, hexPart, numVectorValues, numCopies,
          initialVectorValue);
stk::io::set_field_output_type(variableSizeField, stk::io::FieldOutputType::VECTOR_3D);

std::string meshSpec = "0,1,TET_4, 1,2,3,4, tetElementPart\n"
                       "0,2,HEX_8, 5,6,7,8,9,10,11,12, hexElementPart";
stk::unit_test_util::setup_text_mesh(*bulkPtr, meshSpec);

stk::mesh::Entity tetElem = bulkPtr->get_entity(stk::topology::ELEM_RANK, 1);
stk::mesh::Entity hexElem = bulkPtr->get_entity(stk::topology::ELEM_RANK, 2);

const int tensorScalarsPerTet = stk::mesh::field_scalars_per_entity(tensorField, tetElem);
const int tensorScalarsPerHex = stk::mesh::field_scalars_per_entity(tensorField, hexElem);
EXPECT_EQ(tensorScalarsPerTet, numTensorValues*numCopies);
EXPECT_EQ(tensorScalarsPerHex, numTensorValues*numCopies);

const int tensorExtent0PerTet = stk::mesh::field_extent0_per_entity(tensorField, tetElem);
const int tensorExtent0PerHex = stk::mesh::field_extent0_per_entity(tensorField, hexElem);
EXPECT_EQ(tensorExtent0PerTet, numTensorValues);
EXPECT_EQ(tensorExtent0PerHex, numTensorValues);

const int tensorExtent1PerTet = stk::mesh::field_extent1_per_entity(tensorField, tetElem);
const int tensorExtent1PerHex = stk::mesh::field_extent1_per_entity(tensorField, hexElem);
EXPECT_EQ(tensorExtent1PerTet, numCopies);
EXPECT_EQ(tensorExtent1PerHex, numCopies);

double* tensorData = stk::mesh::field_data(tensorField, hexElem);
for (int i = 0; i < tensorScalarsPerHex; ++i) {
  EXPECT_EQ(initialTensorValue[i], tensorData[i]);
}

const int vectorScalarsPerTet = stk::mesh::field_scalars_per_entity(variableSizeField,
          tetElem);
const int vectorScalarsPerHex = stk::mesh::field_scalars_per_entity(variableSizeField,
          hexElem);
EXPECT_EQ(vectorScalarsPerTet, numVectorValues);
EXPECT_EQ(vectorScalarsPerHex, numVectorValues*numCopies);

const int vectorExtent0PerTet = stk::mesh::field_extent0_per_entity(variableSizeField,
          tetElem);
const int vectorExtent0PerHex = stk::mesh::field_extent0_per_entity(variableSizeField,
          hexElem);
EXPECT_EQ(vectorExtent0PerTet, numVectorValues);
EXPECT_EQ(vectorExtent0PerHex, numVectorValues);

const int vectorExtent1PerTet = stk::mesh::field_extent1_per_entity(variableSizeField,
          tetElem);
const int vectorExtent1PerHex = stk::mesh::field_extent1_per_entity(variableSizeField,
          hexElem);
EXPECT_EQ(vectorExtent1PerTet, 1);
EXPECT_EQ(vectorExtent1PerHex, numCopies);

double* vectorTetData = stk::mesh::field_data(variableSizeField, tetElem);
for (int i = 0; i < vectorScalarsPerTet; ++i) {
  EXPECT_EQ(initialVectorValue[i], vectorTetData[i]);
```

```
  }

  double* vectorHexData = stk::mesh::field_data(variableSizeField, hexElem);
  for (int i = 0; i < vectorScalarsPerHex; ++i) {
    EXPECT_EQ(initialVectorValue[i], vectorHexData[i]);
  }
}
```

## 4.10. STK Multi-State Fields

Some application time-stepping algorithms use multi-state fields to assist with separating and updating the field values for time-step $n$, $n-1$, $n$ 1, etc. STK Mesh supports fields with up to 6 states.

**Listing 4.38 Examples of multi-state field usage**
**code/stk/stk_doc_tests/stk_mesh/useMultistateFields.cpp**

```
TEST(stkMeshHowTo, useMultistateField)
{
  const unsigned spatialDimension = 3;
  stk::mesh::MeshBuilder builder(MPI_COMM_WORLD);
  builder.set_spatial_dimension(spatialDimension);
  builder.set_entity_rank_names(stk::mesh::entity_rank_names());
  std::shared_ptr<stk::mesh::BulkData> bulkPtr = builder.create();
  stk::mesh::MetaData& metaData = bulkPtr->mesh_meta_data();

  typedef stk::mesh::Field<double> ScalarField;
  const unsigned numStates = 2;
  ScalarField& temperatureFieldStateNp1 =
            metaData.declare_field<double>(stk::topology::NODE_RANK, "temperature",
            numStates);

  double initialTemperatureValue = 1.0;
  stk::mesh::put_field_on_entire_mesh_with_initial_value(temperatureFieldStateNp1,
            &initialTemperatureValue);

  metaData.commit();
  stk::mesh::BulkData& mesh = *bulkPtr;
  mesh.modification_begin();
  stk::mesh::EntityId nodeId = 1;
  stk::mesh::Entity node = mesh.declare_node(nodeId);
  mesh.modification_end();

  EXPECT_EQ(stk::mesh::StateNP1, temperatureFieldStateNp1.state());
  double* temperatureStateNp1 = stk::mesh::field_data(temperatureFieldStateNp1, node);
  EXPECT_EQ(initialTemperatureValue, *temperatureStateNp1);
  double newTemperatureValue = 2.0;
  *temperatureStateNp1 = newTemperatureValue;

  ScalarField& temperatureFieldStateN =
            temperatureFieldStateNp1.field_of_state(stk::mesh::StateN);
  double* temperatureStateN = stk::mesh::field_data(temperatureFieldStateN, node);
  EXPECT_EQ(initialTemperatureValue, *temperatureStateN);

  mesh.update_field_data_states();

  temperatureStateN = stk::mesh::field_data(temperatureFieldStateN, node);
  EXPECT_EQ(newTemperatureValue, *temperatureStateN);
}
```

### 4.11. STK Field-BLAS

STK provides several functions that implement BLAS operations on Fields. Here is the current list of Field BLAS functions:

- `field_axpy`
- `field_axpby`
- `field_product`
- `field_copy`
- `field_dot`
- `field_scale`
- `field_fill`
- `field_swap`
- `field_nrm2`
- `field_asum`
- `field_amax`
- `field_amin`

The next snippet illustrates the usage of a couple of these functions (`field_fill` and `field_axpy`). Note that each function has an overload that takes a `Selector`, to facilitate operating on only a selected subset of the field values. For simplicity, this example uses the overload that doesn't require a `Selector`.

**Listing 4.39 Example of using STK Field-BLAS**
**code/stk/stk_doc_tests/stk_mesh/useFieldBLAS.cpp**

```
TEST(stkMeshHowTo, useFieldBLAS)
{
  if (stk::parallel_machine_size(MPI_COMM_WORLD) > 1) { GTEST_SKIP(); }
  stk::mesh::MeshBuilder builder(MPI_COMM_WORLD);
  builder.set_spatial_dimension(SpatialDimension);
  std::unique_ptr<stk::mesh::BulkData> bulkPtr = builder.create();
  stk::mesh::MetaData& metaData = bulkPtr->mesh_meta_data();

  typedef stk::mesh::Field<double> DoubleField;
  DoubleField& pressureField = metaData.declare_field<double>(stk::topology::ELEM_RANK,
           "pressure");
  DoubleField& displacementsField = metaData.declare_field<double>(stk::topology::NODE_RANK,
           "displacements");
  DoubleField& velocityField = metaData.declare_field<double>(stk::topology::NODE_RANK,
           "velocity");

  double initialPressureValue = 4.4;
  constexpr unsigned numValuesPerNode = 3;
  stk::mesh::put_field_on_entire_mesh_with_initial_value(pressureField,
           &initialPressureValue);
  stk::mesh::put_field_on_mesh(displacementsField, metaData.universal_part(),
           numValuesPerNode, nullptr);
  stk::mesh::put_field_on_mesh(velocityField, metaData.universal_part(), numValuesPerNode,
           nullptr);

```

```
  create_two_tet_element_mesh(*bulkPtr);

  //incompatible fields, elem-rank vs node-rank
  EXPECT_ANY_THROW(stk::mesh::field_copy(pressureField, displacementsField));

  stk::mesh::field_fill(99.0, displacementsField);
  stk::mesh::field_fill(10.0, velocityField);
  const double alpha = 5.0;
  stk::mesh::field_axpy(alpha, displacementsField, velocityField);

  const double expectedVal = 10.0 + alpha*99.0;

  auto expectEqualVal = [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
    const double* velocityDataForNode = stk::mesh::field_data(velocityField, node);
    for(unsigned i=0; i<numValuesPerNode; ++i) {
      EXPECT_NEAR(expectedVal, velocityDataForNode[i], 1.e-8);
    }
  };

  stk::mesh::for_each_entity_run(*bulkPtr, stk::topology::NODE_RANK, expectEqualVal);
}
```

## 4.12. STK Mesh NGP (Running on GPU)

The STK *NGP* classes provide GPU capabilities for STK Mesh based applications.

In STK we commonly use the term *NGP* (Next Generation Platform) to refer to GPU platforms and newer platforms other than "traditional" CPU machines. STK Mesh NGP objects use Kokkos for allocating memory and moving data between host and device spaces portably. STK Mesh has been built and tested on a variety of architectures, including Nvidia Cuda, AMD ROCm/HIP, and SYCL.

The basic work-flow involves creating/initializing an STK Mesh with fields on the CPU host as usual, and then creating `NgpMesh` and `NgpField` references using API functions described below. After the `NgpMesh` and `NgpField` objects have been created, one can write GPU kernels in which mesh traversal and field-data access can be performed on the GPU device.

Applications and algorithms which utilize dynamic mesh modification (creating/deleting entities, changing part membership, ghosting, etc) must continue to use the CPU host for the modification operations, and then request an update for the GPU device. Care has been taken to only update (i.e., copy to the GPU) the modified portion of the mesh and fields where possible, for efficiency.

As a brief aside, note that `NgpMesh` and `NgpField` are aliases, which refer to different concrete types depending on whether the macro `STK_USE_DEVICE_MESH` is defined. If a GPU capability is present, i.e., if `KOKKOS_ENABLE_CUDA` or `KOKKOS_ENABLE_HIP` are defined, then `STK_USE_DEVICE_MESH` is defined. It is also possible to manually define `STK_USE_DEVICE_MESH` in a CPU build for testing and debugging purposes.

| GPU build or `STK_USE_DEVICE_MESH` | CPU build and not `STK_USE_DEVICE_MESH` |
|---|---|
| `NgpMesh = DeviceMesh` | `NgpMesh = HostMesh` |
| `NgpField = DeviceField` | `NgpField = HostField` |

Application code that is written in terms of `NgpMesh` and `NgpField` will run identically whether built for a GPU platform or for a CPU platform. When built for a GPU platform, NgpMesh and NgpField are implemented using Kokkos to manage memory allocation and access on device. When built for a CPU platform, NgpMesh and NgpField are implemented as thin wrappers over the "native" BulkData and Field objects.

### 4.12.1. Example STK Mesh NGP usage

There are a variety of demonstrations of NGP usage in the file stk_doc_tests/stk_mesh/howToNgp.cpp. These tests utilize a mesh testing fixture to minimize duplication of initialization and setup code. Much of the setup is not shown in the following example snippets, but all of the details can be found in the source file.

This first test snippet shows how to obtain an NgpMesh object given a pre-existing, pre-initialized BulkData object using stk::mesh::get_update_ngp_mesh(bulk). After BulkData has been modified, the NgpMesh will need to be updated by calling get_updated_ngp_mesh again.

**Listing 4.40 Example of checking whether NGP mesh is up to date**
**code/stk/stk_doc_tests/stk_mesh/howToNgp.cpp**

```
MeshType& ngpMesh = stk::mesh::get_updated_ngp_mesh(bulk);
EXPECT_TRUE(ngpMesh.is_up_to_date());

bulk.modification_begin();
stk::mesh::Entity node1 = bulk.get_entity(stk::topology::NODE_RANK, 1);
bulk.change_entity_parts(node1, stk::mesh::ConstPartVector{extraPart});
bulk.modification_end();

EXPECT_FALSE(ngpMesh.is_up_to_date());

{
  MeshType& newNgpMesh = stk::mesh::get_updated_ngp_mesh(bulk);
  EXPECT_TRUE(newNgpMesh.is_up_to_date());
}

EXPECT_TRUE(ngpMesh.is_up_to_date());
```

This next snippet demonstrates using for_each_entity_run to traverse the elements of the mesh and retrieving an element's node connectivity. It illustrates that stk::mesh::Entity can be copied from host to device, and their values are the same on host and device for a given instance of STK Mesh.

**Listing 4.41 Example of retrieving and comparing NGP mesh connectivity**
**code/stk/stk_doc_tests/stk_mesh/howToNgp.cpp**

```
stk::mesh::Entity elem1_host = bulk.get_entity(stk::topology::ELEM_RANK, 1);
stk::mesh::ConnectedEntities elem1_nodes_host = bulk.get_connected_entities(elem1_host,
           stk::topology::NODE_RANK);
stk::mesh::Entity node0_host = elem1_nodes_host[0];
stk::mesh::Entity node7_host = elem1_nodes_host[7];

const stk::mesh::NgpMesh & ngpMesh = stk::mesh::get_updated_ngp_mesh(bulk);

stk::mesh::Selector allElems = bulk.mesh_meta_data().universal_part();

stk::mesh::for_each_entity_run(ngpMesh, stk::topology::ELEM_RANK, allElems,
```

```
    KOKKOS_LAMBDA(const stk::mesh::FastMeshIndex& elemIndex) {
      stk::mesh::Entity elem = ngpMesh.get_entity(stk::topology::ELEM_RANK, elemIndex);
      stk::mesh::EntityId elemId = ngpMesh.identifier(elem);
      if (elemId == 1) {
        STK_NGP_ThrowRequire(elem == elem1_host);
        const stk::mesh::NgpMesh::BucketType& ngpBucket =
            ngpMesh.get_bucket(stk::topology::ELEM_RANK, elemIndex.bucket_id);
        STK_NGP_ThrowRequire(ngpBucket.topology() == stk::topology::HEX_8);

        stk::mesh::NgpMesh::ConnectedNodes nodes =
            ngpMesh.get_nodes(stk::topology::ELEM_RANK, elemIndex);
        STK_NGP_ThrowRequire(nodes.size() == ngpBucket.topology().num_nodes());
        STK_NGP_ThrowRequire(node0_host == nodes[0]);
        STK_NGP_ThrowRequire(node7_host == nodes[7]);

        stk::mesh::FastMeshIndex nodeIndex = ngpMesh.fast_mesh_index(nodes[0]);
        stk::mesh::NgpMesh::ConnectedEntities node0_elems =
            ngpMesh.get_elements(stk::topology::NODE_RANK, nodeIndex);
        STK_NGP_ThrowRequire(1 == node0_elems.size());
        STK_NGP_ThrowRequire(node0_elems[0] == elem);
      }
    });
```

Typically an application that uses a GPU will have code units that access field data on the GPU and other code units that access field data on the CPU host. This requires that field data be copied back and forth so that the the most up-to-date data is present in the memory space where it is needed, at the time it is needed. To facilitate this, the field classes (including `FieldBase` and `NgpField`) have methods `sync_to_host()`, `sync_to_device()`, `clear_sync_state()`, `modify_on_host()`, `modify_on_device()`, etc. The `modify` methods set a state flag indicating which memory space the field was most recently modified in. The `sync` methods copy the field's data to the desired memory space if it has been marked as modified in the other memory space and reset the previously mentioned state flag. Note that these are host only methods and cannot be called in device kernels.

The following workflow is recommended when writing device kernels:

1. Ensure needed data is updated on the device by calling `field.sync_to_device()`
2. Access field data in the device kernel using `NgpField<ScalarType>::operator(stk::mesh::FastMeshIndex, int)`
3. Mark modified fields by calling `field.modify_on_device()`

Similarly, the following workflow is recommended when writing code which will always run on host:

1. Ensure needed data is updated on the host by calling `field.sync_to_host()`
2. Access field data using one of the `stk::mesh::field_data` overloads
3. Mark modified fields by calling `field.modify_on_host()`

Synchronization methods are no-ops if the field data is already up-to-date in the target memory space. When fields are going to be over-written without reading their current data, `field.clear_sync_state()` can be used instead of synchronizing.

The next snippet demonstrates setting the values of a field on the GPU device.

**Listing 4.42 Example of setting field values on GPU**
**code/stk/stk_doc_tests/stk_mesh/howToNgp.cpp**

```
stk::mesh::NgpMesh & ngpMesh = stk::mesh::get_updated_ngp_mesh(bulk);
EXPECT_EQ(bulk.mesh_meta_data().spatial_dimension(), ngpMesh.get_spatial_dimension());

stk::mesh::NgpField<double>& ngpMeshField =
          stk::mesh::get_updated_ngp_field<double>(meshField);
EXPECT_EQ(meshField.mesh_meta_data_ordinal(), ngpMeshField.get_ordinal());

ngpMeshField.clear_sync_state();

stk::mesh::for_each_entity_run(ngpMesh, rank, meshPart,
                               KOKKOS_LAMBDA(const stk::mesh::FastMeshIndex& entity)
                               {
                                 ngpMeshField(entity, 0) = fieldVal;
                               });

ngpMeshField.modify_on_device();

```

The next snippet demonstrates retrieving and checking the values of a field on the CPU host.

**Listing 4.43 Example of calling field sync to host**
**code/stk/stk_doc_tests/stk_mesh/howToNgp.cpp**

```
const stk::mesh::MetaData& meta = bulk.mesh_meta_data();
const stk::mesh::BucketVector& buckets = bulk.get_buckets(stk::topology::ELEM_RANK,
          meta.locally_owned_part());

stkField.sync_to_host();

for(const stk::mesh::Bucket* bptr : buckets) {
  for(stk::mesh::Entity elem : *bptr) {
    const double* fieldData = stk::mesh::field_data(stkField, elem);
    EXPECT_EQ(*fieldData, expectedFieldValue);
  }
}

```

### 4.12.2. STK Mesh NGP with Multi-State Fields

The method used to perform state-rotation for multi-state fields is `BulkData::update_field_data_states()`. By default this method only rotates the states of host fields, not device fields. To get correct device field states, it is necessary to first sync to host, then do the state update, then sync back to device. This isn't optimal, because the sync'ing is expensive. An optional boolean argument to the `BulkData::update_field_data_states()` method can cause the device field states to also be rotated, meaning that it is not necessary to do the sync'ing and thus performance is much better. This isn't the default behavior because subtle correctness issues arise if the calling code has persistent value-copies of device-fields. In this case a state-rotation cannot update the underlying `Kokkos::View` objects in the copied device fields and they continue pointing to the previous un-rotated field states. If you must hold value-copies of device fields, it is necessary to do the host-rotation with companion sync/modify calls to get correct behavior.

This next snippet demonstrates using a multi-state field in GPU-capable code, using the approach of sync'ing to get the correct states back to the device.

**Listing 4.44 Example of using multi-state field**
**code/stk/stk_doc_tests/stk_mesh/howToNgpMultiStateFields.cpp**

```
  EXPECT_EQ(stk::mesh::StateNew, stkFieldNew.state());
  stk::mesh::Field<double>& stkFieldOld = stkFieldNew.field_of_state(stk::mesh::StateOld);
  EXPECT_EQ(stk::mesh::StateOld, stkFieldOld.state());

  constexpr double oldValue = 1.0;
  constexpr double newValue = 2.0;
  set_field_on_host(*bulkPtr, stkFieldOld, oldValue);
  set_field_on_host(*bulkPtr, stkFieldNew, newValue);

  stk::mesh::NgpMesh& ngpMesh = stk::mesh::get_updated_ngp_mesh(*bulkPtr);
  stk::mesh::NgpField<double>& ngpFieldOld =
            stk::mesh::get_updated_ngp_field<double>(stkFieldOld);
  stk::mesh::NgpField<double>& ngpFieldNew =
            stk::mesh::get_updated_ngp_field<double>(stkFieldNew);

  check_field_on_device(ngpMesh, ngpFieldOld, oldValue);
  check_field_on_device(ngpMesh, ngpFieldNew, newValue);

  stk::mesh::sync_to_host_and_mark_modified(meta);
  bulkPtr->update_field_data_states();

#ifdef STK_USE_DEVICE_MESH
  check_field_on_device(ngpMesh, ngpFieldOld, oldValue);
  check_field_on_device(ngpMesh, ngpFieldNew, newValue);
#else
  check_field_on_device(ngpMesh, ngpFieldOld, newValue);
  check_field_on_device(ngpMesh, ngpFieldNew, oldValue);
#endif

  ngpFieldOld.sync_to_device();
  ngpFieldNew.sync_to_device();

  check_field_on_device(ngpMesh, ngpFieldOld, newValue);
  check_field_on_device(ngpMesh, ngpFieldNew, oldValue);

```

This next snippet demonstrates using a multi-state field in GPU-capable code, using the boolean argument for rotating the device-field states at the same time the host-rotation is done, resulting in correct states with optimal performance.

**Listing 4.45 Example of using multi-state field**
**code/stk/stk_doc_tests/stk_mesh/howToNgpMultiStateFields.cpp**

```
  stk::mesh::NgpMesh& ngpMesh = stk::mesh::get_updated_ngp_mesh(*bulkPtr);
  stk::mesh::NgpField<double>& ngpFieldOld =
            stk::mesh::get_updated_ngp_field<double>(stkFieldOld);
  stk::mesh::NgpField<double>& ngpFieldNew =
            stk::mesh::get_updated_ngp_field<double>(stkFieldNew);

  constexpr double oldValue = 1.0;
  constexpr double newValue = 2.0;

  set_field_on_device(ngpMesh, ngpFieldOld, oldValue);
  set_field_on_device(ngpMesh, ngpFieldNew, newValue);
#ifdef STK_USE_DEVICE_MESH
  check_field_on_host(*bulkPtr, stkFieldOld, 0.0);
  check_field_on_host(*bulkPtr, stkFieldNew, 0.0);
#else
```

```
  check_field_on_host(*bulkPtr, stkFieldOld, oldValue);
  check_field_on_host(*bulkPtr, stkFieldNew, newValue);
#endif

  const bool rotateNgpFieldViews = true;
  bulkPtr->update_field_data_states(rotateNgpFieldViews);

  check_field_on_device(ngpMesh, ngpFieldOld, newValue);
  check_field_on_device(ngpMesh, ngpFieldNew, oldValue);

```

This page intentionally left blank.

# 5. STK IO

## 5.1. STK IO: usage examples

The STK IO module enables reading from and writing to Exodus [1] files with STK Mesh. STK IO provides a wide range of capabilities. It provides basic reading and writing of mesh data, as well as transient field operations such as incrementally adding time steps to results output files, reading and writing restart files, and history/heartbeat output.

### *5.1.1. Reading mesh data to create a STK Mesh*

The first example illustrates using the free-functions `fill_mesh` and `write_mesh` to populate a STK Mesh object and write it out to an Exodus file. In this case we are giving `fill_mesh` a "generated mesh" specification which causes it to use the SEACAS/IOSS generated mesh capability to create a mesh. This is an easy way to create meshes for specific testing scenarios. `fill_mesh` can also be given a file-name and it will then read the mesh data from an Exodus mesh file.

**Listing 5.1 Filling a mesh using generated-mesh data and writing to an Exodus file**
**code/stk/stk_doc_tests/stk_io/readMesh.cpp**

```
    std::shared_ptr<stk::mesh::BulkData> stkMesh =
            stk::mesh::MeshBuilder(communicator).create();

    //+ Create a basic mesh with a hex block, 3 shell blocks, 3 nodesets, and 3 sidesets.
    const std::string generatedFileName = "generated:8x8x8|shell:xyz|nodeset:xyz|sideset:XYZ";
    stk::io::fill_mesh(generatedFileName, *stkMesh);
    stk::io::write_mesh(mesh_file_name, *stkMesh);

```

The following examples will illustrate usage of the class `stk::io::StkMeshIoBroker`, which is the most general way to access most STK IO capabilities.

The first example illustrates a basic mesh reading scenario, filling a STK Mesh with data from an Exodus file. A STK Part will be created for each element block, nodeset, and sideset in the input mesh file and the name of the corresponding part will be the same as the name of the block or set in the mesh file.

Note that in this case `StkMeshIoBroker` creates the objects `MetaData` and `BulkData`, and this example also demonstrates obtaining these objects as `std::shared_ptr`s. It is also possible to have `StkMeshIoBroker` operate on pre-existing instances of `MetaData` and `BulkData`.

**Listing 5.2 Reading mesh data to create a STK mesh**
**code/stk/stk_doc_tests/stk_io/readMesh.cpp**

```
    // ============================================================
    //+ EXAMPLE:
    //+ Read mesh data from the specified file.
    stk::io::StkMeshIoBroker stkIo(communicator);
    stkIo.add_mesh_database(mesh_file_name, stk::io::READ_MESH);

    //+ Creates meta data; creates parts
    stkIo.create_input_mesh();

    //+ Modifications to the meta data (such as creating extra parts and fields)
    //+ is generally be done here.

    //+ Commit the meta data and create the bulk data.
    //+ Populate the bulk data with data from the mesh file.
    stkIo.populate_bulk_data();

    // ============================================================
    //+ VERIFICATION
    //+ In this case we know that the mesh (specified above) contains
    //+ 4 element blocks, 3 nodesets, and 3 sidesets
    //+ There should be a STK Mesh Part for each of those.
    std::shared_ptr<const stk::mesh::MetaData> meta = stkIo.meta_data_ptr();
    EXPECT_NE(nullptr, meta->get_part("block_1"));
    EXPECT_NE(nullptr, meta->get_part("block_2"));
    EXPECT_NE(nullptr, meta->get_part("block_3"));
    EXPECT_NE(nullptr, meta->get_part("block_4"));

    EXPECT_NE(nullptr, meta->get_part("nodelist_1"));
    EXPECT_NE(nullptr, meta->get_part("nodelist_2"));
    EXPECT_NE(nullptr, meta->get_part("nodelist_3"));

    EXPECT_NE(nullptr, meta->get_part("surface_1"));
    EXPECT_NE(nullptr, meta->get_part("surface_2"));
    EXPECT_NE(nullptr, meta->get_part("surface_3"));

    std::shared_ptr<const stk::mesh::BulkData> bulk = stkIo.bulk_data_ptr();
    stk::mesh::EntityVector shellElems;
    stk::mesh::get_entities(*bulk, stk::topology::ELEM_RANK, *meta->get_part("block_2"),
            shellElems);
    EXPECT_EQ(64u, shellElems.size());

```

### 5.1.2. Reading mesh data to create a STK Mesh allowing StkMeshIoBroker to go out of scope

This example shows how to read mesh data from a file and create a STK Mesh corresponding to that mesh data while also allowing the StkMeshIoBroker to go out of scope without deleting the STK Mesh.

**Listing 5.3 Reading mesh data to create a STK mesh using set bulk data**
**code/stk/stk_doc_tests/stk_mesh/createStkMeshAlt1.cpp**

```
TEST(StkMeshHowTo, CreateStkMesh)
{
  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) != 1) { return; }
  const std::string exodusFileName = "example.exo";

  create_example_exodus_file(communicator, exodusFileName);
  // Creation of STK Mesh objects.
  // MetaData creates the universal_part, locally-owned part, and globally shared part.
```

```
std::shared_ptr<stk::mesh::BulkData> stkMeshBulkDataPtr =
            stk::mesh::MeshBuilder(communicator).create();
stk::mesh::MetaData& stkMeshMetaData = stkMeshBulkDataPtr->mesh_meta_data();

// Read the mesh data from the Exodus file and populate an STK Mesh.
// The order of the following lines in {} are important
{
  stk::io::StkMeshIoBroker exodusFileReader(communicator);

  // Provide STK Mesh object to be populated
  exodusFileReader.set_bulk_data(*stkMeshBulkDataPtr);

  exodusFileReader.add_mesh_database(exodusFileName, stk::io::READ_MESH);

  // Populate the MetaData which has the descriptions of the Parts and Fields.
  exodusFileReader.create_input_mesh();

  // Populate entities in STK Mesh from Exodus file
  exodusFileReader.populate_bulk_data();
}

// Verify that the STK Mesh has 512 elements.
stk::mesh::Selector allEntities = stkMeshMetaData.universal_part();
std::vector<size_t> entityCounts;
stk::mesh::count_entities(allEntities, *stkMeshBulkDataPtr, entityCounts);
EXPECT_EQ(512u, entityCounts[stk::topology::ELEMENT_RANK]);
unlink(exodusFileName.c_str());
}
```

### 5.1.3. *Reading mesh data to create a STK Mesh, delaying field allocations*

This example is almost the same as the previous except it delays the allocation of field data so that the application can modify the mesh. If the field data is allocated prior to the mesh modification, the reordering and moving of field data memory may be expensive; if the field data allocation is delayed, no reordering or moving of memory is needed.

The field data memory allocation delay is accomplished by calling `populate_mesh()` and `populate_field_data()` instead of `populate_bulk_data()`. Any mesh modifications, for example, creating mesh edges or mesh faces is performed prior to calling `populate_field_data()`.

**Listing 5.4 Reading mesh data to create a STK mesh; delay field allocation**
**code/stk/stk_doc_tests/stk_io/readMeshDelayFieldAllocation.cpp**

```
  // ============================================================
  //+ EXAMPLE:
  //+ Read mesh data from the specified file.
  stk::io::StkMeshIoBroker stkIo(communicator);
  stkIo.add_mesh_database(mesh_name, stk::io::READ_MESH);

  //+ Creates meta data; creates parts
  stkIo.create_input_mesh();

  //+ Any modifications to the meta data must be done here.
  //+ This includes declaring fields.

  //+ Commit the meta data and create the bulk data.
  //+ populate the bulk data with data from the mesh file.
  stkIo.populate_mesh();

```

```
//+ Application would call mesh modification here.
//+ for example, create_edges().

//+ Mesh modifications complete, allocate field data.
stkIo.populate_field_data();


```

#### 5.1.3.1. Face creation for input sidesets

**Sidesets on volume elements where no shells are involved**

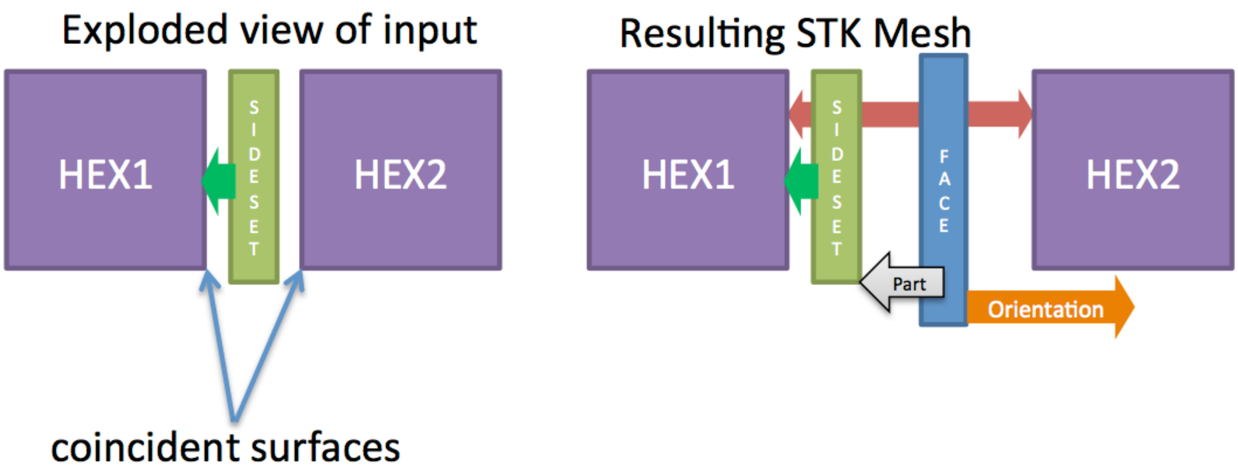



**Figure 5-1. Sideset face creation in STK IO for 2 hexes.**

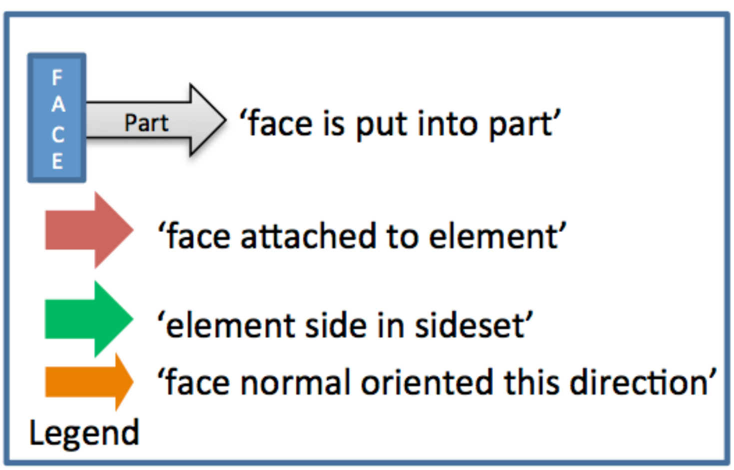



**Figure 5-2. Legend for Sideset Face Creation**

The simple case of reading in Exodus files with sidesets on an exposed or interior surfaces of volume elements (like hexes, tetrahedra, etc.) creates single faces on each surface during mesh read by StkMeshIOBroker. Additional sidesets on exposed or interior surfaces do not create additional faces but do add that face into additional STK parts.

When a face is created due to a sideset in Exodus, it is connected to all elements that share those nodes on a surface. So even if a sideset is present on an interior surface and has only one adjacent volume element, it will be connected to both volume elements that share that interior surface.

This includes doubly-sided sidesets with sides on the two adjacent interior surfaces on neighboring volume elements. In this case, only a single face that is connected to the two neighboring volume elements will be created but it will be added to two STK parts. Whichever side of these coincident sidesets is listed first in the Exodus file will be created first, hence the orientation of that side will be used to set the orientation of the face. The SEACAS utility `ncdump` is useful in determining the ordering of sides and sidesets in Exodus files.

Figure 5-1 shows an example for 2 hexes with a sideset on the leftmost interior surface. Figure 5-2 shows the legend. Listing 5.5 documents the behavior and shows how to check.

**Listing 5.5 Face creation during IO for one sideset between hexes**
**code/stk/stk_doc_tests/stk_mesh/IOSidesetFaceCreation.cpp**

```
TEST(StkMeshHowTo, StkIO2Hex1SidesetFaceCreation)
{
  if (stk::parallel_machine_size(MPI_COMM_WORLD) == 1) {
    //  -------  |S  -------             -------  |F  -------
    //  |     |  |I  |     |             |     |  |A  |     |
    //  |HEX1 5<-|D  4 HEX2| --STK-IO--> |HEX1 5<-|C->4 HEX2|
    //  |     |  |E  |     |             |     |  |E  |     |
    //  -------  |S  -------             -------  |   -------
    //           |E                               |----> face is put into
    //           |T                                       part surface_1
    //                                            |---> orientation points outward
    //                                                   from Hex1 face5

    stk::io::StkMeshIoBroker stkMeshIoBroker(MPI_COMM_WORLD);
    stkMeshIoBroker.add_mesh_database("ALA.e", stk::io::READ_MESH);
    stkMeshIoBroker.create_input_mesh();
    stkMeshIoBroker.populate_bulk_data();

    stk::mesh::BulkData &mesh = stkMeshIoBroker.bulk_data();
    stk::mesh::EntityVector all_faces;
    stk::mesh::get_entities(mesh, stk::topology::FACE_RANK, all_faces);
    std::sort(all_faces.begin(),all_faces.end());
    unsigned expected_num_faces = 1;
    ASSERT_EQ(expected_num_faces, all_faces.size());
    size_t face_index = 0;
    stk::mesh::Entity face = all_faces[face_index];
    stk::topology faceTopology = mesh.bucket(face).topology();
    ASSERT_EQ(stk::topology::QUAD_4, faceTopology);

    EXPECT_TRUE(mesh.bucket(face).member(*mesh.mesh_meta_data().get_part("surface_1")));

    unsigned expected_connected_elements = 2;
    ASSERT_EQ(expected_connected_elements, mesh.num_elements(face));

    const stk::mesh::Entity * connected_elements = mesh.begin_elements(face);
    const stk::mesh::ConnectivityOrdinal * which_side_of_element =
            mesh.begin_element_ordinals(face);
    const stk::mesh::Permutation* face_permutations = mesh.begin_element_permutations(face);

    {
      int element_count = 0;
      stk::mesh::Entity hex_2 = connected_elements[element_count];
      EXPECT_EQ(2u, mesh.identifier(hex_2));
      unsigned expected_face_ordinal = 4;
      EXPECT_EQ(expected_face_ordinal, which_side_of_element[element_count]);
      bool is_positive_permutation =
```

```
            faceTopology.is_positive_polarity(face_permutations[element_count]);
      EXPECT_FALSE(is_positive_permutation);
    }

    {
      int element_count = 1;
      stk::mesh::Entity hex_1 = connected_elements[element_count];
      EXPECT_EQ(1u, mesh.identifier(hex_1));
      unsigned expected_face_ordinal = 5;
      EXPECT_EQ(expected_face_ordinal, which_side_of_element[element_count]);
      bool is_positive_permutation =
            faceTopology.is_positive_polarity(face_permutations[element_count]);
      EXPECT_TRUE(is_positive_permutation);
    }

  }
}
```

### Sidesets on shell elements

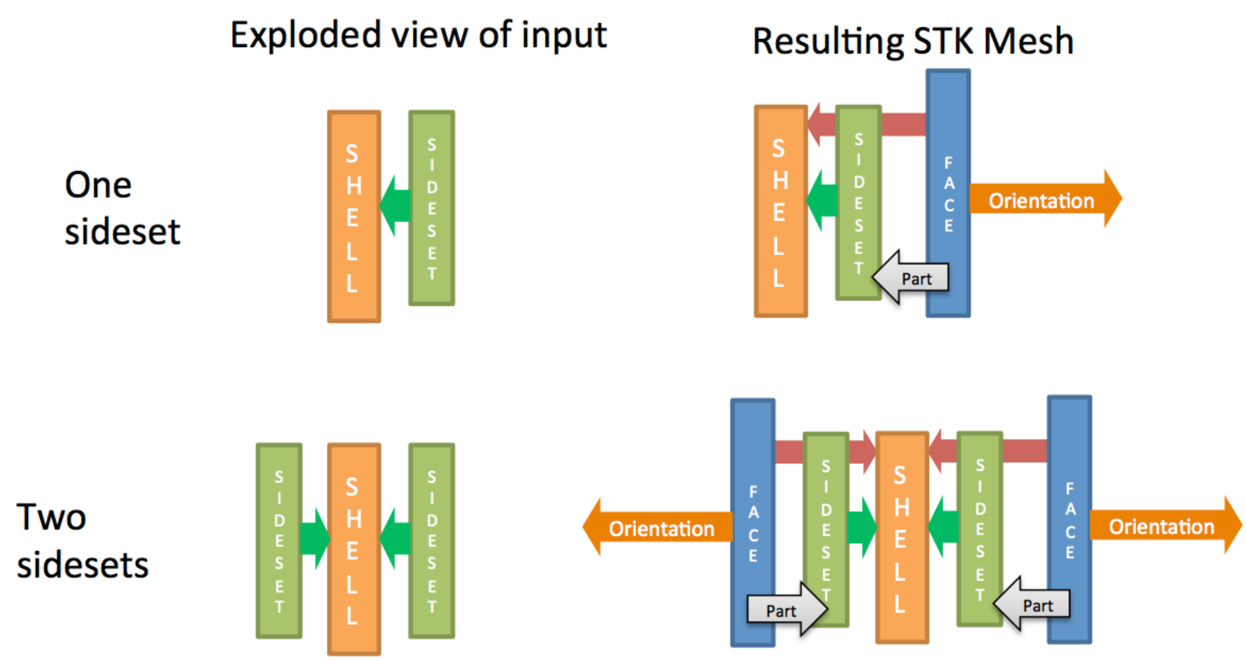



**Figure 5-3. Sideset face creation in STK IO for one shell.**

Sides in sidesets can be created on either surface of a shell or both surfaces. If a single side is present in the Exodus file, a single face will be created and connected to the shell on a single surface. If two sides are present, two faces will be created with opposite permutations and individually connected to single distinct surfaces of the shell.

Figure 5-3 shows an example of two cases on a single shell. Figure 5-2 shows the legend.

### Sidesets on stacked shell elements

On coincident shells, a maximum of two faces are ever created with opposite permutations, no matter how many sidesets are present. Extra sidesets cause parts to be added to the faces. If a single face is created, it is hooked to the same orientation of every coincident shell. If two faces are created, they are individually hooked to the same orientation of all coincident shells.

### Sidesets on mixed volume and shell elements

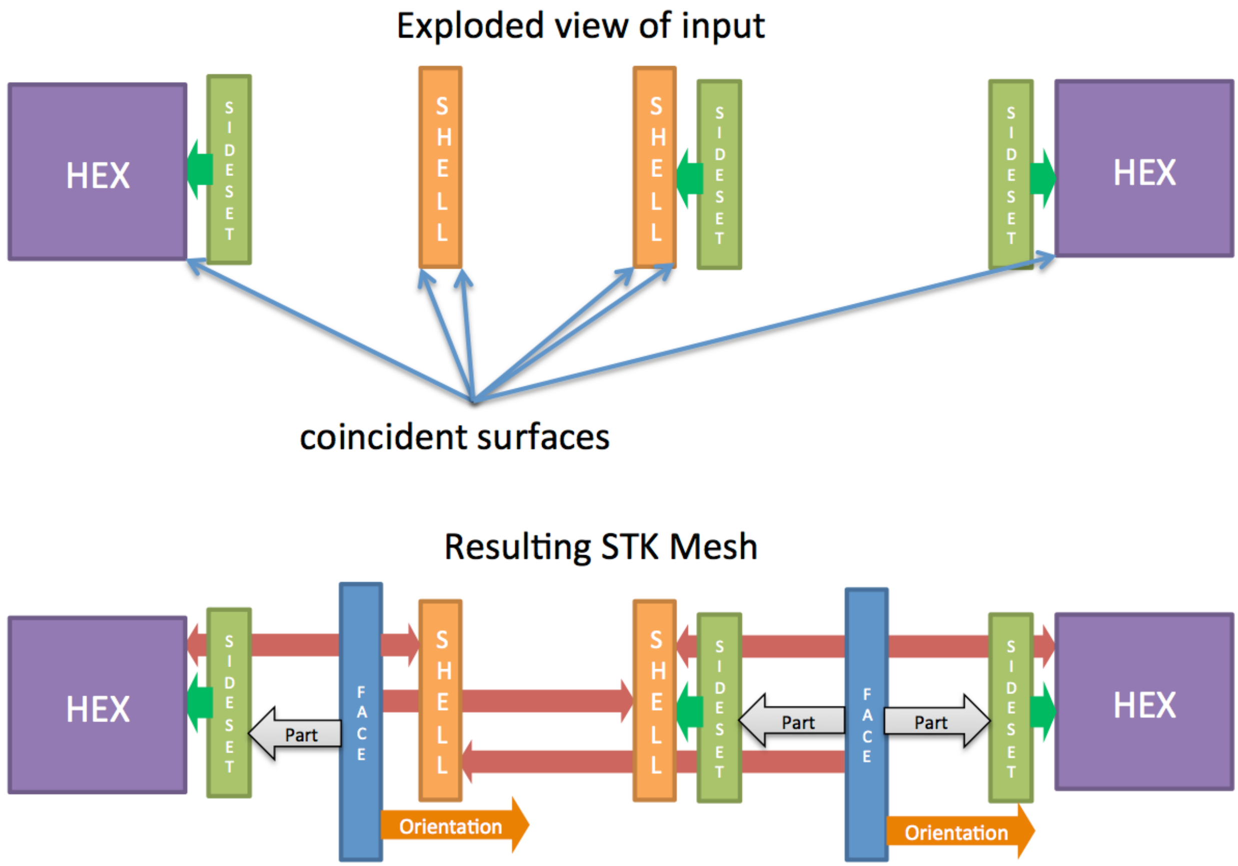



**Figure 5-4. Sideset face creation in STK IO for a complicated example with stacked shells between two hex elements and multiple sidesets.**

When shells are adjacent to volume elements, a maximum of two faces can be created (as opposed to single face with no shells present).

The first side in the first sideset (from the ordering in Exodus as checked by `ncdump`) determines the orientation of the face created for this surface on the element. If this side is on a volume element, it will be hooked to the opposite orientation of any and all coincident shells. If this side is on a shell element, it will be hooked to the same orientation of all other coincident shells but the opposite orientation of any adjacent surfaces on volume elements. If additional sides in sidesets are present in Exodus that would create faces that are already defined, additional parts will be created but not additional faces. If additional sides in sidesets would create a face on the opposing orientation of the shell, then it will be created and hooked to all other shell elements on that orientation and the opposite orientation of any adjacent surfaces on volume elements. Note that orientations of faces on volume elements are always outward directed.

Figure 5-4 an example of two shells between two hexes with three sidesets, only two faces are created. Figure 5-2 shows the legend. Listing 5.6 shows relevant code for checking the ordinals, permutations and parts.

Listing 5.6 Face creation during IO for shells between hexes with sidesets
code/stk/stk_doc_tests/stk_mesh/IOSidesetFaceCreation.cpp

```
TEST(StkMeshHowTo, StkIO2Hex2Shell3SidesetFaceCreation)
{
  if (stk::parallel_machine_size(MPI_COMM_WORLD) == 1) {
    //  -------  |S |S| |S|  |S |S  -------
    //  |     |  |I |H| |H|  |I |I  |     |
    //  |HEX1 5<-|D |E| |E0<-|D |D->4 HEX2|
    //  |     |  |E |L| |L|  |E |E  |     | |
    //  -------  |S |L| |L|  |S |S  ------- |
    //           |E  3   4   |E |E          |
    //           |T          |T |T          STK
    //                                      IO
    //                                       |
    //                                       V
    //
    //  -------  |F  |S|  |S|             |F  -------
    //  |     |  |A--|H|->1H|             |A  |     |
    //  |HEX1 5<-|C->1E|  |E0<------------|C->4 HEX2|
    //  |     |  |E  |L0<-|L|-------------|E  |     |
    //  -------  |   |L|  |L|             |   -------
    //           |    3    4              |
    //           |---> orientation        |-->orientation
    //           |---> in surface_1 part  |-->in surface_2 and
    //                                         surface_3 parts


    stk::io::StkMeshIoBroker stkMeshIoBroker(MPI_COMM_WORLD);
    stkMeshIoBroker.add_mesh_database("ALefLRA.e", stk::io::READ_MESH);
    stkMeshIoBroker.create_input_mesh();
    stkMeshIoBroker.populate_bulk_data();

    stk::mesh::BulkData &mesh = stkMeshIoBroker.bulk_data();
    stk::mesh::EntityVector all_faces;
    stk::mesh::get_entities(mesh, stk::topology::FACE_RANK, all_faces);
    std::sort(all_faces.begin(),all_faces.end());
    unsigned expected_num_faces = 2;
    ASSERT_EQ(expected_num_faces, all_faces.size());

    stk::topology faceTopology = mesh.bucket(all_faces[0]).topology();
    ASSERT_EQ(stk::topology::QUAD_4, faceTopology);
    ASSERT_EQ(faceTopology, mesh.bucket(all_faces[1]).topology());

    size_t face_index = 0;
    {
      stk::mesh::Entity face = all_faces[face_index];
      unsigned expected_connected_elements = 3;
      ASSERT_EQ(expected_connected_elements, mesh.num_elements(face));

      EXPECT_TRUE(mesh.bucket(face).member(*mesh.mesh_meta_data().get_part("surface_1")));

      const stk::mesh::Entity * connected_elements = mesh.begin_elements(face);
      const stk::mesh::ConnectivityOrdinal * which_side_of_element =
            mesh.begin_element_ordinals(face);
      const stk::mesh::Permutation* face_permutations = mesh.begin_element_permutations(face);

      {
        int element_count = 0;
        stk::mesh::Entity shell_3 = connected_elements[element_count];
        EXPECT_EQ(3u, mesh.identifier(shell_3));
        unsigned expected_face_ordinal = 1;
        EXPECT_EQ(expected_face_ordinal, which_side_of_element[element_count]);
        bool is_positive_permutation =
             faceTopology.is_positive_polarity(face_permutations[element_count]);
        EXPECT_FALSE(is_positive_permutation);
      }
      {
```

```
        int element_count = 1;
        stk::mesh::Entity shell_4 = connected_elements[element_count];
        EXPECT_EQ(4u, mesh.identifier(shell_4));
        unsigned expected_face_ordinal = 1;
        EXPECT_EQ(expected_face_ordinal, which_side_of_element[element_count]);
        bool is_positive_permutation =
             faceTopology.is_positive_polarity(face_permutations[element_count]);
        EXPECT_FALSE(is_positive_permutation);
      }
      {
        int element_count = 2;
        stk::mesh::Entity hex_1 = connected_elements[element_count];
        EXPECT_EQ(1u, mesh.identifier(hex_1));
        unsigned expected_face_ordinal = 5;
        EXPECT_EQ(expected_face_ordinal, which_side_of_element[element_count]);
        bool is_positive_permutation =
             faceTopology.is_positive_polarity(face_permutations[element_count]);
        EXPECT_TRUE(is_positive_permutation);
      }
    }

    face_index = 1;
    {
      stk::mesh::Entity face = all_faces[face_index];
      unsigned expected_connected_elements = 3;
      ASSERT_EQ(expected_connected_elements, mesh.num_elements(face));

      EXPECT_TRUE(mesh.bucket(face).member(*mesh.mesh_meta_data().get_part("surface_2")));
      EXPECT_TRUE(mesh.bucket(face).member(*mesh.mesh_meta_data().get_part("surface_3")));

      const stk::mesh::Entity * connected_elements = mesh.begin_elements(face);
      const stk::mesh::ConnectivityOrdinal * which_side_of_element =
             mesh.begin_element_ordinals(face);
      const stk::mesh::Permutation* face_permutations = mesh.begin_element_permutations(face);

      {
        int element_count = 0;
        stk::mesh::Entity shell_3 = connected_elements[element_count];
        EXPECT_EQ(3u, mesh.identifier(shell_3));
        unsigned expected_face_ordinal = 0;
        EXPECT_EQ(expected_face_ordinal, which_side_of_element[element_count]);
        bool is_positive_permutation =
             faceTopology.is_positive_polarity(face_permutations[element_count]);
        EXPECT_FALSE(is_positive_permutation);
      }
      {
        int element_count = 1;
        stk::mesh::Entity shell_4 = connected_elements[element_count];
        EXPECT_EQ(4u, mesh.identifier(shell_4));
        unsigned expected_face_ordinal = 0;
        EXPECT_EQ(expected_face_ordinal, which_side_of_element[element_count]);
        bool is_positive_permutation =
             faceTopology.is_positive_polarity(face_permutations[element_count]);
        EXPECT_FALSE(is_positive_permutation);
      }
      {
        int element_count = 2;
        stk::mesh::Entity hex_2 = connected_elements[element_count];
        EXPECT_EQ(2u, mesh.identifier(hex_2));
        unsigned expected_face_ordinal = 4;
        EXPECT_EQ(expected_face_ordinal, which_side_of_element[element_count]);
        bool is_positive_permutation =
             faceTopology.is_positive_polarity(face_permutations[element_count]);
        EXPECT_TRUE(is_positive_permutation);
      }
    }
  }
}
```

### 5.1.4. Outputting STK Mesh

**Listing 5.7 Writing a STK Mesh**
**code/stk/stk_doc_tests/stk_io/howToWriteMesh.cpp**

```
#include <gtest/gtest.h>
#include <stk_unit_test_utils/getOption.h>
#include <unistd.h>

#include <stk_io/StkMeshIoBroker.hpp>
#include <stk_io/WriteMesh.hpp>
#include <stk_mesh/base/BulkData.hpp>
#include <stk_mesh/base/MeshBuilder.hpp>
#include <stk_mesh/base/Comm.hpp>
#include <stk_mesh/base/MetaData.hpp>
#include <stk_unit_test_utils/CommandLineArgs.hpp>
#include <stk_unit_test_utils/ioUtils.hpp>

namespace
{

TEST(StkIoHowTo, WriteMesh)
{
  std::string filename = "output.exo";
  {
    std::shared_ptr<stk::mesh::BulkData> bulk =
             stk::mesh::MeshBuilder(MPI_COMM_WORLD).create();
    stk::io::fill_mesh("generated:1x1x4", *bulk);

    stk::io::StkMeshIoBroker stkIo;
    stkIo.set_bulk_data(bulk);
    size_t outputFileIndex = stkIo.create_output_mesh(filename, stk::io::WRITE_RESULTS);
    stkIo.write_output_mesh(outputFileIndex);
    stkIo.write_defined_output_fields(outputFileIndex);
  }

  {
    std::shared_ptr<stk::mesh::BulkData> bulk =
             stk::mesh::MeshBuilder(MPI_COMM_WORLD).create();
    stk::io::fill_mesh(filename, *bulk);

    std::vector<size_t> entityCounts;
    stk::mesh::comm_mesh_counts(*bulk, entityCounts);
    EXPECT_EQ(4u, entityCounts[stk::topology::ELEM_RANK]);
  }

  unlink(filename.c_str());
}

TEST(StkIoHowTo, generateMeshWith64BitIds)
{
  std::string meshSpec = stk::unit_test_util::get_option("-i", "1x1x4");
  std::string fullMeshSpec = "generated:"+meshSpec;

  std::string filename = "output.exo";
  stk::io::StkMeshIoBroker inputBroker;

  //+ Set properties to ensure that 64-bit integers will be used
  inputBroker.property_add(Ioss::Property("INTEGER_SIZE_API" , 8));
  inputBroker.property_add(Ioss::Property("INTEGER_SIZE_DB" , 8));
  std::shared_ptr<stk::mesh::BulkData> bulk = stk::mesh::MeshBuilder(MPI_COMM_WORLD).create();
  stk::io::fill_mesh_preexisting(inputBroker, fullMeshSpec, *bulk);
```

```
  stk::io::write_mesh_with_large_ids_and_fields(filename, *bulk);

  if (stk::unit_test_util::GlobalCommandLineArguments::self().get_argc() == 0) {
    unlink(filename.c_str());
  }
}

}
```

### 5.1.5. Outputting STK Mesh With Internal Sidesets

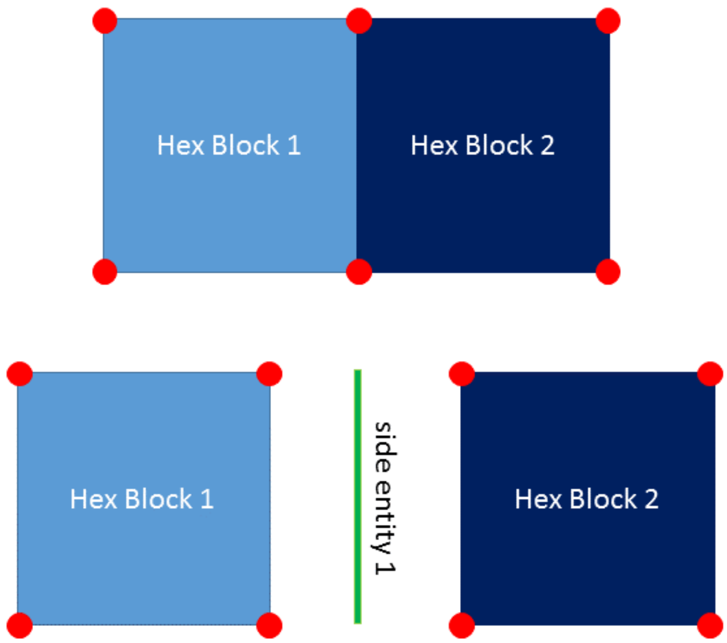



**Figure 5-5. Example mesh used for Listing 5.8**

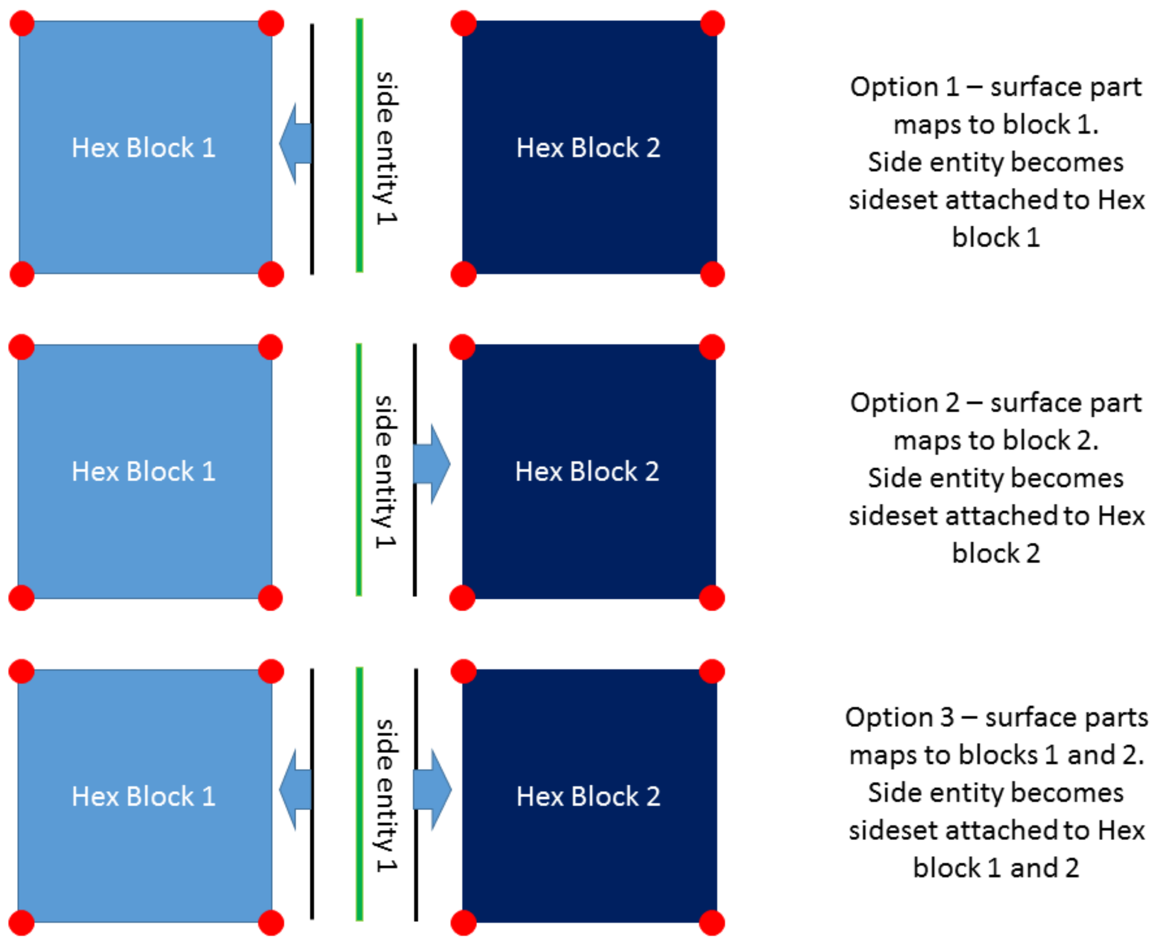



**Figure 5-5. Options for creating a sideset for Listing 5.8**

**Listing 5.8 Writing a STK Mesh**
**code/stk/stk_doc_tests/stk_io/howToWriteMeshWithInternalSidesets.cpp**

```
std::vector<const stk::mesh::Part*> blocks;
for(const std::string& blockName : testData.blockNames)
{
  stk::mesh::Part *block = meta.get_part(blockName);
  blocks.push_back(block);
}

meta.set_surface_to_block_mapping(&sideSetPart, blocks);

```

### 5.1.6. Outputting results data from a STK Mesh

This example shows how an application can output calculated field data to a results database.

**Listing 5.9 Writing calculated field data to a results database**
**code/stk/stk_doc_tests/stk_io/writeResults.cpp**

```
82      // ============================================================
83      //+ EXAMPLE:
84      //+ Read mesh data from the specified file.
85      stk::io::StkMeshIoBroker stkIo(communicator);
86      stkIo.add_mesh_database(mesh_name, stk::io::READ_MESH);
87
88      //+ Creates meta data; creates parts
89      stkIo.create_input_mesh();
90
91      //+ Declare a field
92      //+ NOTE: Fields must be declared before "populate_bulk_data()" is called
93      //+       since it commits the meta data.
94      const std::string fieldName = "disp";
95      stk::mesh::Field<double> &field =
                stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, fieldName, 1);
96      stk::mesh::put_field_on_mesh(field, stkIo.meta_data().universal_part(), nullptr);
97
98      //+ commit the meta data and create the bulk data.
99      //+ populate the bulk data with data from the mesh file.
100     stkIo.populate_bulk_data();
101
102     // ============================================================
103     //+ Create results file. By default, all parts created from the input
104     //+ mesh will be written to the results output file.
105     size_t fh = stkIo.create_output_mesh(results_name, stk::io::WRITE_RESULTS);
106
107     //+ The field will be output to the results file with the default field name.
108     stkIo.add_field(fh, field);
109
110     // Iterate the application's execute loop five times and output
111     // field data each iteration (mimicing time steps).
112     for (int step=0; step < 5; step++) {
113       double time = step;
114
115       //simulate time-varying result field...
116       double value = 10.0 * time;
117       stk::mesh::field_fill(value, field);
118
119       //+ Output the field data calculated by the application.
120       stkIo.begin_output_step(fh, time);
121       stkIo.write_defined_output_fields(fh);
122       stkIo.end_output_step(fh);
123     }
124
```

### 5.1.7. Outputting a field with an alternative name to a results file

The client can specify a field name for results output that is different from the internally used STK Mesh field name. The results output field name is specified as the second argument to the `add_field()` function. The code excerpt shown below replaces line 108 in the previous example (Listing 5.9) to cause the name of the field on the output

**Listing 5.10 Outputting a field with an alternative name**
**code/stk/stk_doc_tests/stk_io/requestedResultsFieldName.cpp**

```
//+ The field 'fieldName' will be output to the results file with the name
        'alternateFieldName'
std::string alternateFieldName("displacement");
stkIo.add_field(fh, field, alternateFieldName);

```

### 5.1.8. Outputting both results and restart data from a STK Mesh

The STK Mesh IO Broker class can output both results data and restart data. Currently, the only difference between results data and restart data is that a restart output will automatically output the multiple states of a multi-state field. If, for example, the application defines a three-state field named "disp", then outputting this field to a restart database will result in the two newest states being output. On the restart database the variables will appear as "disp" and "disp.N." Outputting this field to a results database will only output the data on the newest state as the variable "disp". When the restart database is read back in, the variables will be restored back to the same states that were written.

The example below shows how an application can output both a results and restart database. The example shows both databases being written on each step, but this is not required – each file can specify its own output frequency.

**Listing 5.11 Write results and restart**
**code/stk/stk_doc_tests/stk_io/writeResultsAndRestart.cpp**

```
// ============================================================
//+ EXAMPLE:
//+ Read mesh data from the specified file.
stk::io::StkMeshIoBroker stkIo(communicator);
stkIo.add_mesh_database(mesh_name, stk::io::READ_MESH);

//+ Creates meta data; creates parts
stkIo.create_input_mesh();

//+ Declare a three-state field
//+ NOTE: Fields must be declared before "populate_bulk_data()" is called
//+       since it commits the meta data.
const std::string fieldName = "disp";
stk::mesh::Field<double> &field =
        stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, fieldName, 3);
stk::mesh::put_field_on_mesh(field, stkIo.meta_data().universal_part(), nullptr);

const stk::mesh::Part& block_1 = *stkIo.meta_data().get_part("block_1");
//+ create a two-state field
stk::mesh::Field<double> &fooSubset =
        stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, "fooSubset",
        2);
stk::mesh::put_field_on_mesh(fooSubset, block_1, nullptr);

//+ commit the meta data and create the bulk data.
//+ populate the bulk data with data from the mesh file.
stkIo.populate_bulk_data();

// ============================================================
//+ Create results file. By default, all parts created from the input
//+ mesh will be written to the results output file.
```

```
    size_t results_fh = stkIo.create_output_mesh(results_name, stk::io::WRITE_RESULTS);

    //+ Create restart file. By default, all parts created from the input
    //+ mesh will be written to the results output file.
    size_t restart_fh = stkIo.create_output_mesh(restart_name, stk::io::WRITE_RESTART);

    //+ The field will be output to the results file with the default field name.
    //+ Only the newest state will be output.
    stkIo.add_field(results_fh, field);

    //+ Output the field to the restart database also.
    //+ The two newest states will be output.
    stkIo.add_field(restart_fh, field);
    stkIo.add_field(restart_fh, fooSubset);

    std::vector<stk::mesh::Entity> nodes;
    stk::mesh::get_entities(stkIo.bulk_data(), stk::topology::NODE_RANK, nodes);

    stk::mesh::FieldBase *statedFieldNp1 = field.field_state(stk::mesh::StateNP1);
    stk::mesh::FieldBase *statedFieldN   = field.field_state(stk::mesh::StateN);
    stk::mesh::FieldBase *statedFieldNm1 = field.field_state(stk::mesh::StateNM1);

    // Iterate the application's execute loop five times and output
    // field data each iteration.
    for (int step=0; step < 5; step++) {
      double time = step;

      // Application execution...
      double value = 10.0 * time;
      for(size_t i=0; i<nodes.size(); i++) {
        double *np1_data = static_cast<double*>(stk::mesh::field_data(*statedFieldNp1,
             nodes[i]));
        *np1_data = value;
        double *n_data   = static_cast<double*>(stk::mesh::field_data(*statedFieldN,
             nodes[i]));
        *n_data   = value + 0.1;
        double *nm1_data = static_cast<double*>(stk::mesh::field_data(*statedFieldNm1,
             nodes[i]));
        *nm1_data = value + 0.2;
      }

      //+ Results output...
      stkIo.begin_output_step(results_fh, time);
      stkIo.write_defined_output_fields(results_fh);
      stkIo.end_output_step(results_fh);

      //+ Restart output...
      stkIo.begin_output_step(restart_fh, time);
      stkIo.write_defined_output_fields(restart_fh);
      stkIo.end_output_step(restart_fh);
    }

```

### 5.1.9. Writing multi-state fields to results output file

The previous example showed that a results file will only output the newest state of a multi-state field. However, it is possible to tell a results file to output multiple states from a multi-state field. Each state of the field must be registered individually. Since each state will have the same field name, the `add_field()` call must also specify the name to be used for the variable on the results database in order to get unique names for each state. The example below shows how to output all three states of a multi-state field to a results database.

**Listing 5.12 Writing multi-state field to results output**
**code/stk/stk_doc_tests/stk_io/usingResults.cpp**

```
const std::string fieldName = "disp";
const std::string np1Name = fieldName+"NP1";
const std::string nName   = fieldName+"N";
const std::string nm1Name = fieldName+"Nm1";
{
  // ============================================================
  //+ INITIALIZATION
  const std::string exodusFileName = "generated:1x1x8";
  stk::io::StkMeshIoBroker stkIo(communicator);
  size_t index = stkIo.add_mesh_database(exodusFileName, stk::io::READ_MESH);
  stkIo.set_active_mesh(index);
  stkIo.create_input_mesh();

  //+ Declare a three-state field
  //+ NOTE: Fields must be declared before "populate_bulk_data()" is called
  //+       since it commits the meta data.
  stk::mesh::Field<double> &field =
      stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, fieldName, 3);
  stk::mesh::put_field_on_mesh(field, stkIo.meta_data().universal_part(), nullptr);

  stkIo.populate_bulk_data();

  size_t fh =
      stkIo.create_output_mesh(resultsFilename, stk::io::WRITE_RESULTS);

  // ============================================================
  //+ EXAMPLE
  //+ Output each state of the multi-state field individually to results file
  stk::mesh::FieldBase *statedFieldNp1 = field.field_state(stk::mesh::StateNP1);
  stk::mesh::FieldBase *statedFieldN   = field.field_state(stk::mesh::StateN);
  stk::mesh::FieldBase *statedFieldNm1 = field.field_state(stk::mesh::StateNM1);

  std::vector<stk::mesh::Entity> nodes;
  stk::mesh::get_entities(stkIo.bulk_data(), stk::topology::NODE_RANK, nodes);

  stkIo.add_field(fh, *statedFieldNp1, np1Name);
  stkIo.add_field(fh, *statedFieldN,   nName);
  stkIo.add_field(fh, *statedFieldNm1, nm1Name);

  // Iterate the application's execute loop five times and output
  // field data each iteration.
  for (int step=0; step < 5; step++) {
    double time = step;

    // Application execution...
    // Generate field data... (details omitted)

    //+ Results output...
    stkIo.begin_output_step(fh, time);
    stkIo.write_defined_output_fields(fh);
    stkIo.end_output_step(fh);
  }

```

### 5.1.10. *Writing multiple output files*

The following example shows how to write multiple output files. Although different fields and global variables are written to each file in the example, the same field or global variable can be written to multiple files.

**Listing 5.13 Writing multiple output files**
**code/stk/stk_doc_tests/stk_io/writingMultipleOutputFiles.cpp**

```
    // ============================================================
    //+ EXAMPLE -- Two results output files
    stk::mesh::FieldBase *displacementField =
        meta_data.get_field(stk::topology::NODE_RANK, displacementFieldName);

    //+ For file one, set up results and global variables
    size_t file1Handle = stkIo.create_output_mesh(resultsFilename1,
                                                  stk::io::WRITE_RESULTS);
    stkIo.add_field(file1Handle, *displacementField);
    stkIo.add_global(file1Handle, globalVarNameFile1, Ioss::Field::REAL);

    //+ For file two, set up results and global variables
    size_t file2Handle = stkIo.create_output_mesh(resultsFilename2,
                                                  stk::io::WRITE_RESULTS);
    stkIo.add_field(file2Handle, *displacementField, nameOnOutputFile);
    stk::mesh::FieldBase *velocityField = meta_data.get_field(stk::topology::NODE_RANK,
            velocityFieldName);
    stkIo.add_field(file2Handle, *velocityField);
    stkIo.add_global(file2Handle, globalVarNameFile2, Ioss::Field::REAL);

    //+ Write output
    double time = 0.0;
    stkIo.begin_output_step(file1Handle, time);
    stkIo.write_defined_output_fields(file1Handle);
    stkIo.write_global(file1Handle, globalVarNameFile1, globalVarValue1);
    stkIo.end_output_step(file1Handle);

    stkIo.begin_output_step(file2Handle, time);
    stkIo.write_defined_output_fields(file2Handle);
    stkIo.write_global(file2Handle, globalVarNameFile2, globalVarValue2);
    stkIo.end_output_step(file2Handle);
  }

```

### 5.1.11. *Outputting nodal variables on a subset of the nodes*

By default, a nodal variable is assumed to be defined on all nodes of the mesh. If the variable does not exist on all nodes, then a value of zero will be output for those nodes. If a nodal variable is only defined on a few of the nodes of the mesh, this can increase the size of the mesh file since it is storing much more data than is required. There is an option in STK Mesh IO Broker to handle this case by creating one or more "nodesets" which consist of the nodes of the part or parts where the nodal variable is defined. The name of the nodeset will be the part name suffixed by "_n". For example, if the part is named "fireset", the nodeset corresponding to the nodes of this part will be named "fireset_n".

**Listing 5.14 Using a nodeset variable to output nodal fields defined on only a subset of the mesh**
**code/stk/stk_doc_tests/stk_io/useNodesetDbVarForNodalField.cpp**

```
    // ============================================================
    // INITIALIZATION
    std::string s_elems_per_edge = std::to_string(num_elems_per_edge);

    //+ Create a generated mesh containg hexes and shells.
    std::string input_filename = s_elems_per_edge + "x" +
        s_elems_per_edge + "x" +
        s_elems_per_edge + "|shell:xyzXYZ";

```

```
    stk::io::StkMeshIoBroker stkIo(communicator);
    stkIo.add_mesh_database(input_filename, "generated",
                            stk::io::READ_MESH);
    stkIo.create_input_mesh();

    stk::mesh::MetaData &meta_data = stkIo.meta_data();
    stk::mesh::Field<double> &temperature =
        meta_data.declare_field<double>(stk::topology::NODE_RANK, appFieldName, 1);

    // ============================================================
    //+ Put the temperature field on the nodes of the shell parts.
    const stk::mesh::PartVector &all_parts = meta_data.get_mesh_parts();
    stk::mesh::Selector shell_subset;
    for (size_t i=0; i < all_parts.size(); i++) {
      const stk::mesh::Part *part = all_parts[i];
      stk::topology topo = part->topology();
      if (topo == stk::topology::SHELL_QUAD_4) {
        stk::mesh::put_field_on_mesh(temperature, *part, nullptr);
      }
    }

    stkIo.populate_bulk_data();

    // Create the output...
    size_t fh = stkIo.create_output_mesh(resultsFilename, stk::io::WRITE_RESULTS);

    //+ The "temperature" field will be output on nodesets consisting
    //+ of the nodes of each part the field is defined on.
    stkIo.use_nodeset_for_sideset_nodes_fields(fh, true);
    stkIo.use_nodeset_for_block_nodes_fields(fh, true);
    stkIo.add_field(fh, temperature, dbFieldName);

    // Add three steps to the database
    // For each step, the value of the field is the value 'time'
    for (size_t i=0; i < 3; i++) {
      double time = i;

      stk::mesh::field_fill(time, temperature);

      stkIo.begin_output_step(fh, time);
      stkIo.write_defined_output_fields(fh);
      stkIo.end_output_step(fh);
    }
    // Verification omitted...

```

### 5.1.12. Get number of time steps from a database

**Listing 5.15 get num time steps**
**code/stk/stk_doc_tests/stk_io/howToGetNumTimeSteps.cpp**

```
TEST(ExodusFileWithVariables, queryingFileWithTimeSteps)
{
  std::shared_ptr<stk::mesh::BulkData> meshPtr =
            stk::mesh::MeshBuilder(MPI_COMM_WORLD).create();
  stk::io::StkMeshIoBroker stkIo;
  stkIo.set_bulk_data(meshPtr);
  std::string fileName("generated:2x3x4|variables:element,2|times:1");
  stk::io::fill_mesh_with_fields(fileName, stkIo, *meshPtr);

  const int expectedNumTimeSteps = 1;
  EXPECT_EQ(expectedNumTimeSteps, stkIo.get_num_time_steps());

```

```
  int validTimeStep = 1;
  stkIo.read_defined_input_fields(validTimeStep);

  int invalidTimeStep = 3;
  EXPECT_ANY_THROW(stkIo.read_defined_input_fields(invalidTimeStep));
}
```

#### 5.1.13. Reading sequenced fields from a database

Sequenced fields have the same base name and are numbered sequentially starting with one (field_1, field_2, ..., field_n). They can be read into individual fields or collapsed into a single multi-dimensioned field.

**Listing 5.16 Reading sequenced fields**
**code/stk/stk_doc_tests/stk_io/setOptionToNotCollapseSequencedFields.cpp**

```
TEST_F(MultipleNumberedFieldsWithSameBaseName, whenReading_collapseToSingleStkField)
{
  stk::unit_test_util::create_mesh_with__field_1__field_2__field_3(filename, get_comm());
  read_mesh(filename);
  EXPECT_EQ(1u, get_meta().get_fields(stk::topology::ELEM_RANK).size());
}

TEST_F(MultipleNumberedFieldsWithSameBaseName,
             whenReadingWithoutCollapseOption_threeStkFieldsAreRead)
{
  stk::unit_test_util::create_mesh_with__field_1__field_2__field_3(filename, get_comm());
  stkIo.set_option_to_not_collapse_sequenced_fields();
  read_mesh(filename);
  EXPECT_EQ(3u, get_meta().get_fields(stk::topology::ELEM_RANK).size());
}
```

#### 5.1.14. Reading initial conditions from a field on a mesh database

This example shows how to read data from an input mesh database at a specified time and put the data into a STK Mesh field for use as initial condition data. The name of the field in the database and the name of the STK Mesh field do not match to illustrate how to specify alternate names. The initial portion of the example, which is not shown, creates a mesh with timesteps at times 0.0, 1.0, and 2.0. The database contains a nodal field called “temp” with the same values for each node. The value is the same as the time (0.0, 1.0, and 2.0) for each time step. The example shows how to specify the reading of the field data at a specified time step.

**Listing 5.17 Reading initial condition data from a mesh database**
**code/stk/stk_doc_tests/stk_io/readInitialCondition.cpp**

```
    // ============================================================
    //+ EXAMPLE:
    //+ Read the value of the "temp" field at step 2 and populate
    //+ the nodal field "temperature" for use as an initial condition
    stk::io::StkMeshIoBroker stkIo(communicator);
    size_t index = stkIo.add_mesh_database(ic_name, stk::io::READ_MESH);
    stkIo.set_active_mesh(index);
    stkIo.create_input_mesh();

    stk::mesh::Field<double> &temperature =
        stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, "temperature", 1);
```

```
  stk::mesh::put_field_on_mesh(temperature, stkIo.meta_data().universal_part(), nullptr);
  stkIo.populate_bulk_data();

  //+ The name of the field on the database is "temp"
  stkIo.add_input_field(stk::io::MeshField(temperature, "temp"));

  //+ Read the field values from the database at time 2.0
  stkIo.read_defined_input_fields(2.0);

  // ============================================================
  //+ VERIFICATION
  //+ The value of the field at all nodes should be 2.0
  stk::mesh::for_each_entity_run(stkIo.bulk_data(), stk::topology::NODE_RANK,
      [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
        double *fieldDataForNode = stk::mesh::field_data(temperature, node);
        EXPECT_DOUBLE_EQ(2.0, *fieldDataForNode);
      }
  );

```

### 5.1.15. *Reading initial conditions from a field on a mesh database – apply to a specified subset of mesh parts*

This example is similar to the previous except that the field data read from the mesh database is limited to a subset of the parts in the model. The mesh consists of seven element blocks – one hex block and six shell blocks. The mesh database contains a single field defined on all blocks. In the example, the reading of the field is limited to the six shell element blocks; the field on the hex element block will not be initialized from the data on the mesh database. The `add_subset()` function is where this is specified.

**Listing 5.18 Reading initial condition data from a mesh database**
**code/stk/stk_doc_tests/stk_io/readInitialConditionSubset.cpp**

```
std::string dbFieldNameShell = "ElementBlock_1";
std::string appFieldName = "pressure";

MPI_Comm communicator = MPI_COMM_WORLD;
int numProcs = stk::parallel_machine_size(communicator);
if (numProcs != 1) {
  return;
}

{
  // ============================================================
  // INITIALIZATION
  //+ Create a generated mesh containg hexes and shells with a
  //+ single element variable -- ElementBlock_1
  std::string input_filename = "9x9x9|shell:xyzXYZ|variables:element,1|times:1";

  stk::io::StkMeshIoBroker stkIo(communicator);
  stkIo.add_mesh_database(input_filename, "generated", stk::io::READ_MESH);
  stkIo.create_input_mesh();

  stk::mesh::MetaData &meta_data = stkIo.meta_data();

  // Declare the element "pressure" field...
  stk::mesh::Field<double> &pressure =
      stkIo.meta_data().declare_field<double>(stk::topology::ELEMENT_RANK, appFieldName, 1);

  // "ElementBlock_1" is the name of the element field on the input mesh.
```

```
    stk::io::MeshField mf(pressure, dbFieldNameShell);

    const stk::mesh::PartVector &all_parts = meta_data.get_mesh_parts();
    for (size_t i=0; i < all_parts.size(); i++) {
      const stk::mesh::Part *part = all_parts[i];

      //+ Put the field on all element block parts...
      stk::mesh::put_field_on_mesh(pressure, *part, nullptr);

      stk::topology topo = part->topology();
      if (topo == stk::topology::SHELL_QUAD_4) {

        //+ But only initialize the "pressure" field from mesh data on the shell parts.
        mf.add_subset(*part);
      }
    }

    stkIo.add_input_field(mf);
    stkIo.populate_bulk_data();

    double time = stkIo.get_input_ioss_region()->get_state_time(1);

    //+ Populate the fields with data from the input mesh.
    stkIo.read_defined_input_fields(time);
```

The previous example specified all of the subset parts on a single `MeshField`. It is also possible to specify a separate `MeshField` for each subset part. This is not the most efficient method, but can be used if other modifications of the `MeshField` are needed for each or some of the subset parts.

**Listing 5.19 Reading initial condition data from a mesh database**
**code/stk/stk_doc_tests/stk_io/readInitialConditionMultiSubset.cpp**

```
    // ============================================================
    // INITIALIZATION
    //+ Create a generated mesh containg hexes and shells with a
    //+ single element variable -- pressure
    std::string input_filename = "9x9x9|shell:xyzXYZ|variables:element,1|times:1";

    stk::io::StkMeshIoBroker stkIo(communicator);
    stkIo.add_mesh_database(input_filename, "generated", stk::io::READ_MESH);
    stkIo.create_input_mesh();

    stk::mesh::MetaData &meta_data = stkIo.meta_data();

    // Declare the element "pressure" field...
    stk::mesh::Field<double> &pressure =
        stkIo.meta_data().declare_field<double>(stk::topology::ELEMENT_RANK, appFieldName, 1);

    const stk::mesh::PartVector &all_parts = meta_data.get_mesh_parts();
    for (size_t i=0; i < all_parts.size(); i++) {
      //+ Put the field on all element block parts...
      stk::mesh::put_field_on_mesh(pressure, *all_parts[i], nullptr);
    }

    // This commits BulkData and populates the coordinates, connectivity, mesh...
    stkIo.populate_bulk_data();

    double time = stkIo.get_input_ioss_region()->get_state_time(1);

    //+ Initialize the "pressure" field from mesh data on the shell parts on demand...
    for (size_t i=0; i < all_parts.size(); i++) {
      stk::topology topo = all_parts[i]->topology();
```

```
    if (topo == stk::topology::SHELL_QUAD_4) {

      stk::io::MeshField mf(pressure, dbFieldNameShell);
      mf.set_read_time(time);
      mf.add_subset(*all_parts[i]);
      stkIo.add_input_field(mf);
    }
  }

  //+ Populate any other fields with data from the input mesh.
  //+ This would *not* know about the MeshFields above since
  //+ "add_input_field()" was not called...
  stkIo.read_defined_input_fields(time);



```

The final example in this section shows that the same STK field can be initialized from different database fields on different parts through the use of multiple `MeshFields` with different subsets. In this example, the "pressure" field on the shell element blocks is initialized from one database element variable and the "pressure" field on the non-shell element blocks is initialized from a different database element variable.

**Listing 5.20 Reading initial condition data from a mesh database**
**code/stk/stk_doc_tests/stk_io/readInitialConditionTwoFieldSubset.cpp**

```
std::string dbFieldNameShell = "ElementBlock_1";
std::string dbFieldNameOther = "ElementBlock_2";
std::string appFieldName = "pressure";

MPI_Comm communicator = MPI_COMM_WORLD;
int numProcs = stk::parallel_machine_size(communicator);
if (numProcs != 1) {
  return;
}

{
  // ============================================================
  // INITIALIZATION
  //+ Create a generated mesh containg hexes and shells with two
  //+ element variables -- ElementBlock_1 and ElementBlock_2
  std::string input_filename = "9x9x9|shell:xyzXYZ|variables:element,2|times:1";

  stk::io::StkMeshIoBroker stkIo(communicator);
  stkIo.add_mesh_database(input_filename, "generated", stk::io::READ_MESH);
  stkIo.create_input_mesh();

  stk::mesh::MetaData &meta_data = stkIo.meta_data();

  // Declare the element "pressure" field...
  stk::mesh::Field<double> &pressure =
      stkIo.meta_data().declare_field<double>(stk::topology::ELEMENT_RANK, appFieldName,1);

  stk::io::MeshField mf_shell(pressure, dbFieldNameShell);
  stk::io::MeshField mf_other(pressure, dbFieldNameOther);

  const stk::mesh::PartVector &all_parts = meta_data.get_mesh_parts();
  for (size_t i=0; i < all_parts.size(); i++) {
    const stk::mesh::Part *part = all_parts[i];

    //+ Put the field on all element block parts...
    stk::mesh::put_field_on_mesh(pressure, *part, nullptr);

    stk::topology topo = part->topology();
```

```
  if (topo == stk::topology::SHELL_QUAD_4) {
    //+ The shell blocks will have the pressure field initialized
    //+ from the dbFieldNameShell database variable.
    mf_shell.add_subset(*part);
  }
  else {
    //+ The non-shell blocks will have the pressure field initialized
    //+ from the dbFieldNameOther database variable.
    mf_other.add_subset(*part);
  }
}

stkIo.add_input_field(mf_shell);
stkIo.add_input_field(mf_other);
stkIo.populate_bulk_data();

double time = stkIo.get_input_ioss_region()->get_state_time(1);

//+ Populate the fields with data from the input mesh.
stkIo.read_defined_input_fields(time);
```

### 5.1.16. Reading initial conditions from a field on a mesh database – only read once

This example is the same as the previous example, except that the initial condition field will only be active for a single read. Once data has been read into the field, it is no longer active for subsequent reads. This is specified by calling `set_read_once(true)` on the input field as shown on line 122.

The `read_defined_input_fields()` function is called twice and it is verified that the field data does not change on the second call since the input field is no longer active at that call.

**Listing 5.21 Reading initial condition data from a mesh database one time only**
**code/stk/stk_doc_tests/stk_io/readInitialConditionOnce.cpp**

```
103 // ============================================================
104 //+ EXAMPLE:
105 //+ Read the value of the "temp" field at step 2 and populate
106 //+ the nodal field "temperature" for use as an initial condition
107 //+ The input field should only be active for one 'read_defined_input_fields'
108 //+ call, so verify this by calling the function again at step 3 and
109 //+ then verify that the field values are still those read from step 2.
110 stk::io::StkMeshIoBroker stkIo(communicator);
111 size_t index = stkIo.add_mesh_database(ic_name, stk::io::READ_MESH);
112 stkIo.set_active_mesh(index);
113 stkIo.create_input_mesh();
114
115 stk::mesh::Field<double> &temperature =
116     stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, "temperature", 1);
117 stk::mesh::put_field_on_mesh(temperature, stkIo.meta_data().universal_part(), nullptr);
118 stkIo.populate_bulk_data();
119
120 //+ The name of the field on the database is "temp"
121 stk::io::MeshField input_field(temperature, "temp", stk::io::MeshField::CLOSEST);
122 input_field.set_read_once(true);
123 stkIo.add_input_field(input_field);
124
125 //+ Read the field values from the database at time 2.0
126 //+ Pass in a time of 2.2 to verify that the value returned is
127 //+ from the closest step and not interpolated.
```

```
    stkIo.read_defined_input_fields(2.2);

    // ============================================================
    //+ VERIFICATION
    //+ The value of the field at all nodes should be 2.0
    auto checkVals = [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
        double *fieldDataForNode = stk::mesh::field_data(temperature, node);
        EXPECT_DOUBLE_EQ(2.0, *fieldDataForNode);
      };

    stk::mesh::for_each_entity_run(stkIo.bulk_data(), stk::topology::NODE_RANK, checkVals);

    //+ Call read_defined_input_fields again and verify that the
    //+ input field registration is no longer active
    //+ since it was specified to be "only_read_once()"
    stkIo.read_defined_input_fields(3.0);

    //+ The value of the field at all nodes should still be 2.0
    stk::mesh::for_each_entity_run(stkIo.bulk_data(), stk::topology::NODE_RANK, checkVals);
```

### *5.1.17. Reading initial conditions from a mesh database field at a specified database time*

This example is similar to the previous two examples except that the database time at which the field data is to be read is specified explicitly instead of being equal to the analysis time. This is specified by calling `set_read_time()` on the input field as shown on line 135.

The `read_defined_input_fields()` function is called with an analysis time argument of `1.0`. The "flux" field gets the database field values corresponding to that time, but the "temp" field gets the database field values at the database time (2.0) time at which it is explicitly specified.

**Listing 5.22 Reading initial condition data from a mesh database at a specified time**
**code/stk/stk_doc_tests/stk_io/readInitialConditionSpecifiedTime.cpp**

```
108     // ============================================================
109     //+ EXAMPLE:
110     //+ Register the reading of database fields "temp" and "flux" to
111     //+ populate the stk nodal fields "temperature" and "heat_flux"
112     //+ for use as initial conditionss.
113     //+ Specify that the "temp" field should be read from database
114     //+ time 2.0 no matter what time is specified in the read_defined_input_fields
115     //+ call.
116     //+ The "flux" field will be read at the database time corresponding
117     //+ to the analysis time passed in to read_defined_input_fields.
118
119     stk::io::StkMeshIoBroker stkIo(communicator);
120     size_t index = stkIo.add_mesh_database(ic_name, stk::io::READ_MESH);
121     stkIo.set_active_mesh(index);
122     stkIo.create_input_mesh();
123
124     stk::mesh::Field<double> &temperature =
125         stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, "temperature", 1);
126     stk::mesh::put_field_on_mesh(temperature, stkIo.meta_data().universal_part(), nullptr);
127
128     stk::mesh::Field<double> &heat_flux =
129         stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, "heat_flux", 1);
130     stk::mesh::put_field_on_mesh(heat_flux, stkIo.meta_data().universal_part(), nullptr);
```

```
    stkIo.populate_bulk_data();

    // The name of the field on the database is "temp"
    stk::io::MeshField temp_field(temperature, "temp", stk::io::MeshField::CLOSEST);
    temp_field.set_read_time(2.0);
    stkIo.add_input_field(temp_field);

    // The name of the field on the database is "flux"
    stk::io::MeshField flux_field(heat_flux, "flux", stk::io::MeshField::CLOSEST);
    stkIo.add_input_field(flux_field);

    //+ Read the field values from the database at time 1.0
    //+ The value of "flux" will be the values from database time 1.0
    //+ However, the value of "temp" will be the values from database time 2.0
    stkIo.read_defined_input_fields(1.0);

    // ============================================================
    //+ VERIFICATION
    stk::mesh::for_each_entity_run(stkIo.bulk_data(), stk::topology::NODE_RANK,
      [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
      //+ The value of the "temperature" field at all nodes should be 2.0
      double *fieldDataForNode = stk::mesh::field_data(temperature, node);
      EXPECT_DOUBLE_EQ(2.0, *fieldDataForNode);

      //+ The value of the "heat_flux" field at all nodes should be 1.0
      fieldDataForNode = stk::mesh::field_data(heat_flux, node);
      EXPECT_DOUBLE_EQ(1.0, *fieldDataForNode);
    });

```

### 5.1.18. *Reading field data from a mesh database – interpolating between database times*

This example shows how to read data from an input mesh database at multiple times. The database field values are linearly interpolated if the analysis time does not match an existing database time. The initial portion of the example, which is not shown, creates a mesh with time steps at times 0.0, 1.0, and 2.0. The database contains a nodal field called "temp" with the same values for each node. The value is the same as the time (0.0, 1.0, and 2.0) for each time step. The example shows how to specify the reading of the field data at multiple steps and linearly interpolating the database data to the specified analysis times. Line 120 shows how to specify that the field data are to be linear interpolated.

**Listing 5.23 Linearly interpolating field data from a mesh database**
**code/stk/stk_doc_tests/stk_io/interpolateNodalField.cpp**

```
101     // ============================================================
102     //+ EXAMPLE:
103     //+ The input mesh database has 3 timesteps with times 0.0, 1.0, 2.0,
104     //+ The value of the field "temp" is equal to the time
105     //+ Read the "temp" value at times 0.0 to 2.0 with an interval
106     //+ of 0.1 (0.0, 0.1, 0.2, 0.3, ..., 2.0) and verify that
107     //+ the field contains the correct interpolated value.
108     stk::io::StkMeshIoBroker stkIo(communicator);
109     stkIo.add_mesh_database(ic_name, stk::io::READ_MESH);
110     stkIo.create_input_mesh();
111
112     stk::mesh::Field<double> &temperature =
113         stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, "temperature", 1);
114     stk::mesh::put_field_on_mesh(temperature, stkIo.meta_data().universal_part(), nullptr);
```

```
    stkIo.populate_bulk_data();

    //+ Specify that the field data are to be linear interpolated.
    stkIo.add_input_field(stk::io::MeshField(temperature, "temp",
                                             stk::io::MeshField::LINEAR_INTERPOLATION));

    //+ If the same stk field (temperature) is added more than once,
    //+ the first database name and settings will be used.  For example,
    //+ the add_input_field below will be ignored with no error or warning.
    stkIo.add_input_field(stk::io::MeshField(temperature, "temp-again",
                                             stk::io::MeshField::LINEAR_INTERPOLATION));

    for (size_t i=0; i < 21; i++) {
      double time = i/10.0;
      //+ Read the field values from the database and verify that they
      //+ are interpolated correctly.
      stkIo.read_defined_input_fields(time);

      // ============================================================
      //+ VERIFICATION
      // The value of the "temperature" field at all nodes should be 'time'
      stk::mesh::for_each_entity_run(stkIo.bulk_data(), stk::topology::NODE_RANK,
        [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
          double *fieldData = stk::mesh::field_data(temperature, node);
          EXPECT_DOUBLE_EQ(time, *fieldData);
        });
    }

```

### 5.1.19. *Combining restart and interpolation of field data*

This example shows how to specify that an analysis, that is using field interpolation, should be restarted. This requires two input databases: one that contains the restart data and another that contains the field data to be interpolated.

The initial portion of the example, which is not shown, creates a restart database with several nodal and element fields containing three time steps at times 0.0, 1.0, and 2.0. It then also creates a database containing the field values which will be interpolated. This database contains 10 time steps (0.0 to 9.0) with the nodal field "temp". The value of the field at each time step is equal to the database time (0.0 to 9.0).

The `add_mesh_database()` function is called twice – once for each database. Since there are multiple mesh databases, the `set_active_mesh()` function is called to specify which mesh is active for subsequent calls. The fields that are to be read from each database are specified using `add_all_mesh_fields_as_input_fields()` for the restart database and `add_input_field()` for the interpolated field database. Note that the file index for the interpolated field database is passed to the `add_input_field()` since that database is not active at the time of the call.

The example then "restarts" the analysis by setting the restart database as the *active mesh* and reads the restart field data at time 1.0. The active mesh is then switched to the mesh database containing the "temp" field and the analysis is then continued up to time 9.0 with the values for the temperature field being interpolated.

**Listing 5.24 Combining restart and field interpolation**
**code/stk/stk_doc_tests/stk_io/restartInterpolatedField.cpp**

```
    // ============================================================
    //+ EXAMPLE:
    //+ The restart mesh database has 3 timesteps with times 0.0, 1.0, 2.0,
    //+ and several fields.
    //+
    //+ The initial condition database has 10 timesteps with times
    //+ 0.0, 1.0, ..., 9.0 and a nodal variable "temp"
    //+ The value of the field "temp" is equal to the time
    //+
    //+ The example will read the restart database at time 1.0
    //+ and then simulate continuing the analysis at that time
    //+ reading the initial condition data from the other database
    //+ interpolating this data.
    stk::io::StkMeshIoBroker stkIo(communicator);
    size_t ic = stkIo.add_mesh_database(ic_name, stk::io::READ_MESH);
    size_t rs = stkIo.add_mesh_database(rs_name, stk::io::READ_RESTART);

    //+ "Restart" the calculation...
    double time = 1.0;
    stkIo.set_active_mesh(rs);
    stkIo.create_input_mesh();

    stkIo.add_all_mesh_fields_as_input_fields();

    stk::mesh::Field<double> &temperature =
        stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, "temperature", 1);
    stk::mesh::put_field_on_mesh(temperature, stkIo.meta_data().universal_part(), nullptr);

    // The name of the field on the initial condition database is "temp"
    stkIo.add_input_field(ic, stk::io::MeshField(temperature, "temp",
                                                 stk::io::MeshField::LINEAR_INTERPOLATION));
    stkIo.populate_bulk_data();

    std::vector<stk::mesh::Entity> nodes;
    stk::mesh::get_entities(stkIo.bulk_data(), stk::topology::NODE_RANK, nodes);

    //+ Read restart data
    stkIo.read_defined_input_fields(time);

    //+ Switch active mesh to "initial condition" database
    stkIo.set_active_mesh(ic);

    double delta_time = 1.0 / 4.0;
    while (time <= 9.0) {
      //+ Read the field values from the database and verify that they
      //+ are interpolated correctly.
      stkIo.read_defined_input_fields(time);

      // ============================================================
      //+ VERIFICATION
      // The value of the "temperature" field at all nodes should be 'time'
      for(size_t i=0; i<nodes.size(); i++) {
        double *fieldDataForNode = stk::mesh::field_data(temperature, nodes[i]);
        EXPECT_DOUBLE_EQ(time, *fieldDataForNode);
      }
      time += delta_time;
    }

```

### 5.1.20. *Interpolating field data from a mesh database with only a single database time*

If an application specifies that the mesh database field data should be linearly interpolated, but the mesh database only has a single time step, then the field data will not be interpolated and instead, the values read from that single time will be used.

The initial portion of the example, which is not shown, creates a mesh with a time step at time 1.0. The database contains a nodal field called "temp" with the same values for each node. The value is the same as the time (1.0).

The example specifies that the field data should be linearly interpolated and then reads the data at multiple steps. Since there is only a single step on the mesh database, all field values are equal to the database values at that step.

**Listing 5.25 Linearly interpolating field data from a mesh database with only a single step**
**code/stk/stk_doc_tests/stk_io/interpolateSingleStep.cpp**

```
// ============================================================
//+ EXAMPLE:
//+ The input mesh database has 1 timesteps with time 1.0
//+ The value of the field "temp" is equal to the time
//+ Read the "temp" value at times 0.0 to 2.0 with an interval
//+ of 0.1 (0.0, 0.1, 0.2, 0.3, ..., 2.0) and verify that
//+ the field value does not change since there are not
//+ enough steps to do any interpolation.
//+
stk::io::StkMeshIoBroker stkIo(communicator);
stkIo.add_mesh_database(ic_name, stk::io::READ_MESH);
stkIo.create_input_mesh();

stk::mesh::Field<double> &temperature =
    stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK,"temperature",1);
stk::mesh::put_field_on_mesh(temperature, stkIo.meta_data().universal_part(), nullptr);

// The name of the field on the database is "temp"
stkIo.add_input_field(stk::io::MeshField(temperature, "temp",
                                         stk::io::MeshField::LINEAR_INTERPOLATION));

stkIo.populate_bulk_data();

for (size_t i=0; i < 21; i++) {
  double time = i/10.0;
  //+ Read the field values from the database and verify that they
  //+ are interpolated correctly.
  stkIo.read_defined_input_fields(time);

  // ============================================================
  //+ VERIFICATION
  // The value of the "temperature" field at all nodes should be 1.0
  stk::mesh::for_each_entity_run(stkIo.bulk_data(), stk::topology::NODE_RANK,
    [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
      double *fieldData = stk::mesh::field_data(temperature, node);
      EXPECT_DOUBLE_EQ(1.0, *fieldData);
    });
}
```

### 5.1.21. Interpolating field data from a mesh database when time is outside database time interval

If an application specifies that the mesh database field data should be linearly interpolated, but requests data at times outside the interval of times present on the mesh database, then the values at the closest database time will be used instead. In other words, the database values are not extrapolated.

The initial portion of the example, which is not shown, creates a mesh with two time steps at times 1.0 and 2.0. The database contains a nodal field called “temp” with the same values for each node. The value is the same as the time (1.0 or 2.0).

The example specifies that the field data should be linearly interpolated and then reads the data at multiple times from 0.0 to 3.0. Since the database only contains data at times 1.0 and 2.0, the field values at times 0.0 to 1.0 will be set to the database values at time 1.0 and the field values at times 2.0 to 3.0 will be set to the database values at time 2.0. The field values at times 1.0 to 2.0 will be linearly interpolated from the database values.

**Listing 5.26 Linearly interpolating field data when the time is outside the database time interval**
**code/stk/stk_doc_tests/stk_io/interpolateOutsideRange.cpp**

```
// ============================================================
//+ EXAMPLE:
//+ The input mesh database has 2 timesteps with time 1.0 and 2.0
//+ The value of the field "temp" is equal to the time
//+ Read the "temp" value at times 0.0 to 3.0 with an interval
//+ of 0.1 (0.0, 0.1, 0.2, 0.3, ..., 2.0).
//+
//+ The times 0.0 to 1.0 and 2.0 to 3.0 are outside
//+ the range of the mesh database so no interpolation
//+ or extrapolation will occur -- the field values
//+ will be set to the values at the nearest time.
//+
//+ Verify that the values from times 0.0 to 1.0
//+ are equal to 1.0 and that the values from 2.0 to 3.0
//+ are equal to 2.0.
//+ The field values from 1.0 to 2.0 will be interpolated
//+
stk::io::StkMeshIoBroker stkIo(communicator);
stkIo.add_mesh_database(ic_name, stk::io::READ_MESH);
stkIo.create_input_mesh();

stk::mesh::Field<double> &temperature =
    stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, "temperature", 1);
stk::mesh::put_field_on_mesh(temperature, stkIo.meta_data().universal_part(), nullptr);

stkIo.populate_bulk_data();

// The name of the field on the database is "temp"
stkIo.add_input_field(stk::io::MeshField(temperature, "temp",
                                         stk::io::MeshField::LINEAR_INTERPOLATION));

for (size_t i=0; i < 21; i++) {
  double time = i/10.0;
  //+ Read the field values from the database and verify that they
  //+ are interpolated correctly.
  stkIo.read_defined_input_fields(time);

  // ============================================================
  //+ VERIFICATION

```

```
    double expected_value = time;
    if (time <= 1.0)
      expected_value = 1.0;
    if (time >= 2.0)
      expected_value = 2.0;

    stk::mesh::for_each_entity_run(stkIo.bulk_data(), stk::topology::NODE_RANK,
      [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
        const double *fieldData = stk::mesh::field_data(temperature, node);
        EXPECT_DOUBLE_EQ(expected_value, *fieldData);
    });
  }

```

### 5.1.22. Error condition – reading initial conditions from a field that does not exist on a mesh database

This example shows the behavior when the application specifies that initial condition or restart data should be read from the input database, but one or more of the specified fields do not exist on the database. The application specifies that the data for the field “displacement” is to be populated from the database field “disp”, which does not exist. Two scenarios are possible. In the first, the application passes in a vector which on return from the `read_defined_input_fields()` function will contain a list of all fields that were not found, with one entry for each missing field state. In the second, the vector is omitted in the call to `read_defined_input_fields()`; in this case, the code will print an error message and throw an exception if there are any fields not found.

**Listing 5.27 Specifying initial conditions from a non-existent field**
**code/stk/stk_doc_tests/stk_io/handleMissingFieldOnRead.cpp**

```
    // ============================================================
    //+ EXAMPLE:
    //+ Demonstrate what happens when application requests the
    //+ reading of a field that does not exist on the input
    //+ mesh database.  The nodal field "displacement" is
    //+ requested for input from the database field "disp" which
    //+ does not exist.
    stk::io::StkMeshIoBroker stkIo(communicator);
    size_t index = stkIo.add_mesh_database(ic_name, stk::io::READ_MESH);
    stkIo.set_active_mesh(index);
    stkIo.create_input_mesh();

    stk::mesh::Field<double> &temperature =
        stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, "temperature", 1);
    stk::mesh::put_field_on_mesh(temperature, stkIo.meta_data().universal_part(), nullptr);

    stk::mesh::Field<double> &displacement =
        stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, "displacement",
          3);
    stk::mesh::put_field_on_mesh(displacement, stkIo.meta_data().universal_part(), nullptr);
    stkIo.populate_bulk_data();

    // The name of the field on the database is "temp"
    // This field does exist and should be read correctly
    stkIo.add_input_field(stk::io::MeshField(temperature, "temp"));

    //+ The name of the field on the database is "disp"
    //+ This field does not exist and will not be found.
    stkIo.add_input_field(stk::io::MeshField(displacement, "disp"));
```

```
    //+ Read the field values from the database at time 2.0
    //+ The 'missing_fields' vector will contain the names of
    //+ any fields that were not found.
    std::vector<stk::io::MeshField> missing_fields;
    stkIo.read_defined_input_fields(2.0, &missing_fields);

    // =============================================================
    //+ VERIFICATION
    //+ The 'missing' vector should be of size 1 and contain
    //+ 'disp'
    EXPECT_EQ(2u, missing_fields.size());
    EXPECT_EQ("disp", missing_fields[0].db_name());
    EXPECT_EQ("displacement", missing_fields[0].field()->name());
    EXPECT_EQ("disp", missing_fields[1].db_name());
    EXPECT_EQ("displacement_STKFS_N", missing_fields[1].field()->name());

    // The value of the "temperature" field at all nodes should be 2.0
    stk::mesh::for_each_entity_run(stkIo.bulk_data(), stk::topology::NODE_RANK,
      [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
        double *fieldDataForNode = stk::mesh::field_data(temperature, node);
        EXPECT_DOUBLE_EQ(2.0, *fieldDataForNode);
      });
```

This example is the same as the previous except that instead of passing in the vector to hold the missing fields, the application will throw an exception for the missing field. Note that if the application throws an exception, it will not read any field data even for the fields that do exist.

**Listing 5.28 Specifying initial conditions from a non-existent field**
**code/stk/stk_doc_tests/stk_io/handleMissingFieldOnReadThrow.cpp**

```
  //+ If read the fields, but don't pass in the 'missing_fields'
  //+ vector, the code will print an error message and throw an
  //+ exception if it doesn't find all of the requested fields.
  EXPECT_ANY_THROW(stkIo.read_defined_input_fields(2.0));

  //+ If code throws due to missing field(s), it will NOT read
  //+ even the fields that exist.
```

### *5.1.23. Interpolation of fields on database with negative times*

Although it is not common, there are occasions when an analysis will use negative times. For example, an analysis may run from time -3.0 to 0.0 to "preload" a structure and then continue from time 0.0 onward to analyze the preloaded structure. This example shows that the field interpolation capability works correctly when the mesh database and the analysis use negative times.

**Listing 5.29 Interpolating fields on a database with negative times**
**code/stk/stk_doc_tests/stk_io/interpolateFieldNegativeTime.cpp**

```
  // =============================================================
  //+ EXAMPLE:
  //+ The input mesh database has 3 timesteps with times -2.0, -1.0, 0.0.
  //+ The value of the field "temp" is equal to the time
  //+ Read the "temp" value at times -2.0 to 0.0 with an interval
  //+ of 0.1 (-2.0, -1.9, -1.8, ..., 0.0) and verify that
```

```
  //+ the field contains the correct interpolated value.
  stk::io::StkMeshIoBroker stkIo(communicator);
  stkIo.add_mesh_database(ic_name, stk::io::READ_MESH);
  stkIo.create_input_mesh();

  stk::mesh::Field<double> &temperature =
      stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, "temperature", 1);
  stk::mesh::put_field_on_mesh(temperature, stkIo.meta_data().universal_part(), nullptr);

  stkIo.populate_bulk_data();

  // The name of the field on the database is "temp"
  stkIo.add_input_field(stk::io::MeshField(temperature, "temp",
                                           stk::io::MeshField::LINEAR_INTERPOLATION));

  for (int i=-20; i <= 0; i++) {
    double time = i/10.0;
    //+ Read the field values from the database and verify that they
    //+ are interpolated correctly.
    stkIo.read_defined_input_fields(time);

    // ============================================================
    //+ VERIFICATION
    // The value of the "temperature" field at all nodes should be 'time'
    stk::mesh::for_each_entity_run(stkIo.bulk_data(), stk::topology::NODE_RANK,
      [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
        double *fieldData = stk::mesh::field_data(temperature, node);
        EXPECT_DOUBLE_EQ(time, *fieldData);
      });
  }

```

### 5.1.24. Interpolation of fields on database with non-monotonically increasing times

In some cases, the database from which the field values are being interpolated may contain non-monotonically increasing time values. For example, the time steps could contain the values 2.0 at step 1, 0.0 at step 2, and 1.0 at step 3. The example shows that the field interpolation capability works correctly in this case.

**Listing 5.30 Interpolating fields on a database with non-monotonically increasing times**
**code/stk/stk_doc_tests/stk_io/interpolateFieldNonMonotonicTime.cpp**

```
  // ============================================================
  //+ EXAMPLE:
  //+ The input mesh database has 3 timesteps with times 2.0, 0.0, 1.0
  //+ which are non-monotonically increasing.
  //+ The value of the field "temp" is equal to the time
  //+ Read the "temp" value at times 0.0 to 2.0 with an interval
  //+ of 0.1 (0.0, 0.1, 0.2, ..., 2.0) and verify that
  //+ the field contains the correct interpolated value.
  stk::io::StkMeshIoBroker stkIo(communicator);
  stkIo.add_mesh_database(ic_name, stk::io::READ_MESH);
  stkIo.create_input_mesh();

  stk::mesh::Field<double> &temperature =
      stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, "temperature", 1);
  stk::mesh::put_field_on_mesh(temperature, stkIo.meta_data().universal_part(), nullptr);

  stkIo.populate_bulk_data();

  // The name of the field on the database is "temp"
  stkIo.add_input_field(stk::io::MeshField(temperature, "temp",
```

```
                                              stk::io::MeshField::LINEAR_INTERPOLATION));

    for (int i=0; i < 21; i++) {
      double time = i/10.0;
      //+ Read the field values from the database and verify that they
      //+ are interpolated correctly.
      stkIo.read_defined_input_fields(time);

      // ============================================================
      //+ VERIFICATION
      // The value of the "temperature" field at all nodes should be 'time'
      stk::mesh::for_each_entity_run(stkIo.bulk_data(), stk::topology::NODE_RANK,
        [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
          double *fieldData = stk::mesh::field_data(temperature, node);
          EXPECT_DOUBLE_EQ(time, *fieldData);
      });
    }

```

### 5.1.25. *Arbitrary analysis time to database time mapping during field input*

There are instances in which the analysis times do not exactly correspond to the times on the mesh database. An example is a mesh database with times in microseconds and the analysis using seconds for the time units. Another example is when the conditions specified on the mesh database describe a cyclic loading over a small time period, but the analysis time runs over multiples of this period.

The `InputFile` class in STK Mesh IO Broker module contains several options for mapping the analysis time to the database time. These include: offset, scale, period, startup, period type, start time, and stop time.

To describe the mapping from analysis time to database time we will use the following notation:

- a variable of type $t_x$ is in units of time.
- $t_{app}$ is application time.
- $t_{db}$ is database time, which is the time that will be used to query the database.
- $t_{period}$ is the length of the cyclic period; it is 0.0 if not cyclic.
- *scale* is the time scaling factor.
- $t_{offset}$ is the time offset.
- The cyclic behavior can either by specified as *CYCLIC* or *REVERSING*. In the cyclic case, the time would repeat as 1,2,3,1,2,3,...; the reversing case would repeat as 1,2,3,2,1,2,3,,.... Both of these have a $t_{period}$ of length 2.

We now describe the mapping:

- If: $t_{app} < t_{start}$ or $t_{app} > t_{stop}$ Then the field is inactive.
- If: $t_{app} < t_{startup}$ Then $t_{db} = t_{app}$.
- Else if cyclic behavior is *CYCLIC* Then $t_{db} = t_{startup} \mod t_{app} - t_{startup}, t_{period}$.

- Else if cyclic behavior is *REVERSING* Then
  - Let $t_{pm} = \mod t_{app} - t_{startup}, 2 \times t_{period}$
  - If: $t_{pm} \leq t_{period}$ Then $t_{db} = t_{startup}\ t_{pm}$
  - Else: $t_{db} = t_{startup}\ 2 \times t_{period} - t_{pm}$.
- Finally: $t_{db} = t_{db} \times scale\ t_{offset}$.

The example below shows an input mesh database containing a nodal field named “temp”. The database contains 3 steps with times 0.0, 10.0, and 20.0; the value of the field at each time is equal to the time value (0.0, 10.0, or 20.0).

The analysis wants to use the data on this mesh to provide linearly interpolated values for the analysis field “temperature”. The mesh database values will be defined as `REVERSING` cyclic with a period length of 2.0; in addition, the times will be scaled by 10. This should result in a mapping of application time ($t_{app}$) to database time ($t_{db}$) of:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $t_{app}$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| $t_{db}$ | 0 | 10 | 20 | 10 | 0 | 10 | 20 | 10 | 0 | 10 | 20 |

**Listing 5.31 Arbitrary analysis time to database time mapping during field input**
**code/stk/stk_doc_tests/stk_io/interpolateFieldCyclic.cpp**

```
// ============================================================
//+ EXAMPLE:
//+ The input mesh database has 3 timesteps with times 0.0, 10.0, 20.0,
//+ The value of the field "temp" is equal to the time
//+ Read the "temp" value at times 0.0 to 10.0 with an interval
//+ of 0.25 (0.0, 0.25, 0.50, 0.75, ..., 10.0)
//+ The mapping from analysis time (0.0 to 10.0) to database
//+ time will be reverse cyclic and scaled.
//+
//+ The parameters are:
//+ * period = 2.0
//+ * scale = 10.0
//+ * offset = 0.0
//+ * cycle type = REVERSING
//+
//+ Analysis Time and DB_Time:
//+ 0  1  2  3   4   5   6   7   8   9  10
//+ 0 10 20 10   0  10  20  10   0  10  20
//+

stk::io::StkMeshIoBroker stkIo(communicator);
size_t idx = stkIo.add_mesh_database(ic_name, stk::io::READ_MESH);
stkIo.create_input_mesh();

stk::mesh::Field<double> &temperature =
    stkIo.meta_data().declare_field<double>(stk::topology::NODE_RANK, "temperature", 1);
stk::mesh::put_field_on_mesh(temperature, stkIo.meta_data().universal_part(), nullptr);

stkIo.populate_bulk_data();

// The name of the field on the database is "temp"
stkIo.add_input_field(stk::io::MeshField(temperature, "temp",
                                          stk::io::MeshField::LINEAR_INTERPOLATION));

//+ Set the periodic parameters on the input mesh...
double period_length = 2.0;
double startup = 0.0;
```

```
double scale = 10.0;
stkIo.get_mesh_database(idx)
    .set_periodic_time(period_length, startup, stk::io::InputFile::REVERSING)
    .set_scale_time(scale)
    .set_start_time(0.0).set_offset_time(0.0).set_stop_time(999.0); // These are optional
double delta_time = 0.25;
double time = 0.0;
double expected = 0.0;
double exp_inc = 10.0 * delta_time;

while (time <= 10.0) {

  //+ Read the field values from the database and verify that they
  //+ are interpolated correctly.
  stkIo.read_defined_input_fields(time);

  // ============================================================
  //+ VERIFICATION
  // The value of the "temperature" field at all nodes should be 'expected'
  stk::mesh::for_each_entity_run(stkIo.bulk_data(), stk::topology::NODE_RANK,
    [&](const stk::mesh::BulkData& bulk, stk::mesh::Entity node) {
    double *fieldData = stk::mesh::field_data(temperature, node);
    EXPECT_DOUBLE_EQ(expected, *fieldData);
  });

  time += delta_time;
  expected += exp_inc;
  if (expected >= 20.0 || expected <= 0.0) {
    exp_inc = -exp_inc;
  }
}

```

### *5.1.26. Error condition – specifying interpolation for an integer field*

This example shows the behavior when the application specifies that linear interpolation should be used for an integer field. Although there are a few instances in which this could be valid, it is not supported and an exception will be thrown when the field is registered.

**Listing 5.32 Error condition – specifying interpolation of an integer field**
**code/stk/stk_doc_tests/stk_io/interpolateIntegerFieldInvalid.cpp**

```
// ============================================================
//+ EXAMPLE:
//+ Interpolated fields cannot be of type integer.
//+ An exception will be thrown if you try to register an
//+ integer interpolated field.

stk::io::StkMeshIoBroker stkIo(communicator);

const std::string generatedFileName = "generated:8x8x8|nodeset:xyz";
stkIo.add_mesh_database(generatedFileName, stk::io::READ_MESH);
stkIo.create_input_mesh();

stk::mesh::Field<int> &integer_field =
    stkIo.meta_data().declare_field<int>(stk::topology::NODE_RANK, "int_field", 1);
stk::mesh::put_field_on_mesh(integer_field, stkIo.meta_data().universal_part(), nullptr);
stkIo.populate_bulk_data();

EXPECT_ANY_THROW(stkIo.add_input_field(stk::io::MeshField(integer_field, "int_field",
        stk::io::MeshField::LINEAR_INTERPOLATION)));

```

### 5.1.27. Working with element attributes

**Listing 5.33 Working with element attributes**
**code/stk/stk_doc_tests/stk_io/readAttributes.cpp**

```
std::vector<double> get_attributes_of_first_element(const stk::mesh::BulkData &bulk, const stk::mesh::Part *ioPart)
{
  stk::mesh::FieldVector attributeFields = get_attribute_fields_for_part(bulk.mesh_meta_data(), ioPart);

  stk::mesh::EntityVector elements;
  stk::mesh::get_entities(bulk, stk::topology::ELEM_RANK, *ioPart, elements);

  std::vector<double> attributes;
  if(!elements.empty()) {
    for(const stk::mesh::FieldBase *field : attributeFields) {
      unsigned numAttribute = stk::mesh::field_scalars_per_entity(*field, elements[0]);
      double *dataForElement = static_cast<double*> (stk::mesh::field_data(*field, elements[0]));
      for(unsigned i=0; i<numAttribute; ++i)
        attributes.push_back(dataForElement[i]);
    }
  }
  return attributes;
}

TEST_F(ExodusFileWithAttributes, readAttributes_haveFieldsWithAttributes)
{
  setup_mesh("hex_spider.exo", stk::mesh::BulkData::AUTO_AURA);

  const stk::mesh::Part *partBlock2  = get_meta().get_part("block_2");
  const stk::mesh::Part *partBlock10 = get_meta().get_part("block_10");

  EXPECT_EQ(1u, get_attributes_of_first_element(get_bulk(), partBlock2).size());
  EXPECT_EQ(7u, get_attributes_of_first_element(get_bulk(), partBlock10).size());
}

void mark_field_as_attribute(stk::mesh::FieldBase &field)
{
  stk::io::set_field_role(field, Ioss::Field::ATTRIBUTE);
}

TEST_F(ExodusFileWithAttributes, addAttribute_haveFieldsWithAttribute)
{
  allocate_bulk(stk::mesh::BulkData::AUTO_AURA);

  stk::io::StkMeshIoBroker stkIo;
  stkIo.set_bulk_data(get_bulk());
  stkIo.add_mesh_database("hex_spider.exo", stk::io::READ_MESH);
  stkIo.create_input_mesh();

  double initialValue = 0.0;
  auto &newAttrField = get_meta().declare_field<double>(stk::topology::ELEM_RANK, "newAttr");
  mark_field_as_attribute(newAttrField);

  const stk::mesh::Part *partBlock10 = get_meta().get_part("block_10");
  stk::mesh::put_field_on_mesh(newAttrField, *partBlock10, &initialValue);

  stkIo.populate_bulk_data();

  EXPECT_EQ(8u, get_attributes_of_first_element(get_bulk(), partBlock10).size());
}
```

### 5.1.28. *Create an output mesh with a subset of the mesh parts*

If a results file that only contains a portion or subset of the parts existing in the STK Mesh is wanted, this can be specified by creating a `Selector` (see Section 4.5) containing the desired output parts and then calling the `set_subset_selector()` function with that `Selector` as an argument. This is illustrated in the following example.

**Listing 5.34 Creating output mesh containing a subset of the mesh parts**
**code/stk/stk_doc_tests/stk_io/subsettingOutputDB.cpp**

```
// ============================================================
// INITIALIZATION
std::string s_elems_per_edge = std::to_string(num_elems_per_edge);

//+ Create a generated mesh containg hexes and shells.
std::string input_filename = s_elems_per_edge + "x" +
    s_elems_per_edge + "x" +
    s_elems_per_edge + "|shell:xyzXYZ";

stk::io::StkMeshIoBroker stkIo(communicator);
size_t index = stkIo.add_mesh_database(input_filename, "generated", stk::io::READ_MESH);
stkIo.set_active_mesh(index);
stkIo.create_input_mesh();
stkIo.populate_bulk_data();

stk::mesh::MetaData &meta_data = stkIo.meta_data();
const stk::mesh::PartVector &all_parts = meta_data.get_mesh_parts();

// ============================================================
//+ EXAMPLE
//+ Create a selector containing just the shell parts.
stk::mesh::Selector shell_subset;
for (size_t i=0; i < all_parts.size(); i++) {
  const stk::mesh::Part *part = all_parts[i];
  stk::topology topo = part->topology();
  if (topo == stk::topology::SHELL_QUAD_4) {
    shell_subset |= *part;
  }
}

// Create the output...
size_t fh = stkIo.create_output_mesh(resultsFilename,
                                     stk::io::WRITE_RESULTS);

//+ Specify that only the subset of parts selected by the
//+ "shell_subset" selector will be on the output database.
stkIo.set_subset_selector(fh, shell_subset);
stkIo.write_output_mesh(fh);
// Verification omitted...
```

### 5.1.29. *Writing and reading global variables*

The following example shows the use of global variables for a scalar double precision floating point value, but a similar interface exists for working with vectors of global values. The example also shows two methods for handling the error condition of accessing a nonexistent global variable.

**Listing 5.35 Writing and reading a global variable**
**code/stk/stk_doc_tests/stk_io/writingAndReadingGlobalVariables.cpp**

```
TEST(StkMeshIoBrokerHowTo, writeAndReadGlobalVariables)
{
  MPI_Comm communicator = MPI_COMM_WORLD;
  int numProcs = stk::parallel_machine_size(communicator);
  if (numProcs != 1) { return; }

  const std::string restartFileName = "OneGlobalDouble.restart";
  const std::string timeStepVarName = "timeStep";
  const double timeStepSize = 1e-6;
  const double currentTime = 1.0;

  //+ Write restart file with time step size as a global variable
  {
    stk::io::StkMeshIoBroker stkIo(communicator);
    const std::string exodusFileName = "generated:1x1x8";
    stkIo.add_mesh_database(exodusFileName, stk::io::READ_MESH);
    stkIo.create_input_mesh();
    stkIo.populate_bulk_data();

    size_t fileIndex =
        stkIo.create_output_mesh(restartFileName, stk::io::WRITE_RESTART);
    stkIo.add_global(fileIndex, timeStepVarName, Ioss::Field::REAL);
    stkIo.begin_output_step(fileIndex, currentTime);
    stkIo.write_global(fileIndex, timeStepVarName, timeStepSize);
    stkIo.end_output_step(fileIndex);
  }

  //+ Read restart file with time step size as a global variable
  {
    stk::io::StkMeshIoBroker stkIo(communicator);
    stkIo.add_mesh_database(restartFileName, stk::io::READ_RESTART);
    stkIo.create_input_mesh();
    stkIo.populate_bulk_data();
    stkIo.read_defined_input_fields(currentTime);
    std::vector<std::string> globalNamesOnFile;
    stkIo.get_global_variable_names(globalNamesOnFile);

    ASSERT_EQ(1u, globalNamesOnFile.size());
    EXPECT_STRCASEEQ(timeStepVarName.c_str(),
                     globalNamesOnFile[0].c_str());
    double timeStepSizeReadFromFile = 0.0;
    stkIo.get_global(globalNamesOnFile[0], timeStepSizeReadFromFile);

    const double epsilon = std::numeric_limits<double>::epsilon();
    EXPECT_NEAR(timeStepSize, timeStepSizeReadFromFile, epsilon);

    //+ If try to get a global that does not exist, will throw
    //+ an exception by default...
    double value = 0.0;
    EXPECT_ANY_THROW(stkIo.get_global("does_not_exist", value));

    //+ If the application wants to handle the error instead (without a try/catch),
    //+ can pass in an optional boolean:
    bool abort_if_not_found = false;
    bool found = stkIo.get_global("does_not_exist", value, abort_if_not_found);
    ASSERT_FALSE(found);
  }

  unlink(restartFileName.c_str());
}
```

### 5.1.30. Writing and reading global parameters

The following example shows the use of `stk::util::Parameter` objects for global variable output and input. The example defines several parameters of type double, integer, vector of doubles, and a vector of integers. The list containing these parameters is iterated and each is defined to be an output global variable. Then, each variable is written in the time step loop. At the end of writing, the file is reopened for reading and the parameter values are restored and checked to make sure the correct values were read.

**Listing 5.36 Writing and reading parameters as global variables**
**code/stk/stk_doc_tests/stk_io/writingAndReadingGlobalParameters.cpp**

```
TEST(StkMeshIoBrokerHowTo, writeAndReadGlobalParameters)
{
  // ============================================================
  //+ INITIALIZATION
  const std::string file_name = "GlobalParameters.e";
  MPI_Comm communicator = MPI_COMM_WORLD;

  // Add some parameters to write and read...
  stk::util::ParameterList params;
  params.set_param("PI", 3.14159);   // Double
  params.set_param("Answer", 42);    // Integer

  std::vector<double> my_vector = { 2.78, 5.30, 6.21 };
  params.set_param("doubles", my_vector); // Vector of doubles...

  std::vector<int> ages = { 55, 49, 21, 19};
  params.set_param("Ages", ages);   // Vector of integers...

  {
    stk::io::StkMeshIoBroker stkIo(communicator);
    const std::string exodusFileName = "generated:1x1x8";
    size_t index = stkIo.add_mesh_database(exodusFileName, stk::io::READ_MESH);
    stkIo.set_active_mesh(index);
    stkIo.create_input_mesh();
    stkIo.populate_bulk_data();

    // ============================================================
    //+ EXAMPLE
    //+ Write output file with all parameters in params list...
    size_t idx = stkIo.create_output_mesh(file_name,
                                          stk::io::WRITE_RESTART);

    stk::util::ParameterMapType::const_iterator i  = params.begin();
    stk::util::ParameterMapType::const_iterator ie = params.end();
    for (; i != ie; ++i) {
      const std::string parameterName = (*i).first;
      stk::util::Parameter &param = params.get_param(parameterName);
      stkIo.add_global(idx, parameterName, param);
    }

    stkIo.begin_output_step(idx, 0.0);

    for (i = params.begin(); i != ie; ++i) {
      const std::string parameterName = (*i).first;
      stk::util::Parameter &param = params.get_param(parameterName);
      stkIo.write_global(idx, parameterName, param);
    }

    stkIo.end_output_step(idx);
  }

  {
```

```
102     // ============================================================
103     //+ EXAMPLE
104     //+ Read parameters from file...
105     stk::io::StkMeshIoBroker stkIo(communicator);
106     stkIo.add_mesh_database(file_name, stk::io::READ_MESH);
107     stkIo.create_input_mesh();
108     stkIo.populate_bulk_data();
109 
110     stkIo.read_defined_input_fields(0.0);
111 
112     stk::util::ParameterMapType::const_iterator i = params.begin();
113     stk::util::ParameterMapType::const_iterator ie = params.end();
114     for (; i != ie; ++i) {
115       const std::string parameterName = (*i).first;
116       stk::util::Parameter &param = params.get_param(parameterName);
117       stkIo.get_global(parameterName, param);
118     }
119 
120     // ============================================================
121     //+ VALIDATION
122     stk::util::ParameterList gold_params; // To compare values read
123     gold_params.set_param("PI", 3.14159);        // Double
124     gold_params.set_param("Answer", 42);         // Integer
125     gold_params.set_param("doubles", my_vector); // Vector of doubles
126     gold_params.set_param("Ages", ages);         // Vector of integers...
127 
128     size_t param_count = 0;
129     for (i = params.begin(); i != ie; ++i) {
130       param_count++;
131       const std::string parameterName = (*i).first;
132       stk::util::Parameter &param = params.get_param(parameterName);
133       stk::util::Parameter &gold_parameter =
134           gold_params.get_param(parameterName);
135       validate_parameters_equal_value(param, gold_parameter);
136     }
137 
138     std::vector<std::string> globalNamesOnFile;
139     stkIo.get_global_variable_names(globalNamesOnFile);
140     ASSERT_EQ(param_count, globalNamesOnFile.size());
141   }
142   // ============================================================
143   // CLEAN UP
144   unlink(file_name.c_str());
145 }
```

### 5.1.31. *Writing global variables automatically*

This example is similar to the previous one except that in this case, the global variables are written automatically without calling `write_global()` for each value. The only changes to the previous example are:

- replace the call to `add_global()` with a call to `add_global_ref()`.
- pass a reference to the value as is shown on line 94, and
- replace the code on lines 90 to 98 of the previous example with the call to `process_output_request()` on line 99.

**Listing 5.37 Automatically writing parameters as global variables**
**code/stk/stk_doc_tests/stk_io/writingAndReadingGlobalParametersAuto.cpp**

```
// ... Setup is the same as in the previous example
// Write output file with all parameters in params list...
{
  stk::io::StkMeshIoBroker stkIo(communicator);
  const std::string exodusFileName = "generated:1x1x8";
  size_t input_index = stkIo.add_mesh_database(exodusFileName, stk::io::READ_MESH);
  stkIo.set_active_mesh(input_index);
  stkIo.create_input_mesh();
  stkIo.populate_bulk_data();

  size_t idx = stkIo.create_output_mesh(file_name,
                                       stk::io::WRITE_RESTART);

  stk::util::ParameterMapType::const_iterator i = params.begin();
  stk::util::ParameterMapType::const_iterator iend = params.end();
  for (; i != iend; ++i) {
    const std::string paramName = (*i).first;
    //+ NOTE: Need a reference to the parameter.
    stk::util::Parameter &param = params.get_param(paramName);
    stkIo.add_global_ref(idx, paramName, param);
  }

  //+ All writing of the values is handled automatically,
  //+ do not need to call write_global
  stkIo.process_output_request(idx, 0.0);
}
// ... Reading is the same as in previous example
```

### 5.1.32. Heartbeat output

The Heartbeat periodically outputs user-defined data to either a text or binary (exodus) file. The data are typically defined in `stk::util::Parameter` objects, but raw integer, double, or complex values can also be specified. The format of the heartbeat output is customizable and consists of an optional "legend" followed by one or more lines containing the current value of the registered variables at each time step. The data can be scalars, vectors, tensors, or other composite types consisting of integer, real, or complex values.

The currently defined basic formats for heartbeat output are:

- `CSV` Comma-separated values. The output consists of a header line containing the names of each variable being output. The names are separated by commas. Each data line consists of comma-separated values.
- `TS_CSV` Time-stamped comma-separated values. Similar to the `CSV` format except that each line is preceded by a timestamp showing, by default, the time of day that the line was output in 24-hour format.
- `TEXT` Similar to `CSV` except that tab characters are used to separate the fields instead of commas.
- `TS_TEXT` Similar to `TEXT` except that each line is preceded by a timestamp.
- `SPYHIS` A format that can be plotted by the *spyplot* graphics program.
- `BINARY` The data will be output to an exodus file as global variables. This is sometimes referred to as a "history" file.

The format is specified as the second argument to the `add_heartbeat_output()` command as shown on line 91 in the example below where the `TEXT` format is selected.

The following example shows the basic usage of the heartbeat capability. In the initialization section, the parameters and their values are defined. Note that in addition to scalar values, vectors of values are also supported. The values to be output to the heartbeat file are defined in lines 93 to 104. The values are output at line 112. Note that the application does not have to individually output each value; the heartbeat system does this automatically. The application only has to make sure that the correct value is in the `parameter.value` prior to calling `process_heartbeat_output()`.

**Listing 5.38 Writing global variables to a Heartbeat file**
**code/stk/stk_doc_tests/stk_io/usingHeartbeat.cpp**

```
61   stk::util::ParameterList params;
62
63   {
64     // ============================================================
65     //+ INITIALIZATION...
66     // Add some params to write and read...
67     params.set_param("PI", -3.14159);  // Double
68     params.set_param("Answer", 42);    // Integer
69
70     std::vector<double> my_vector;
71     my_vector.push_back(2.78);
72     my_vector.push_back(5.30);
73     my_vector.push_back(6.21);
74     params.set_param("some_doubles", my_vector);    // Vector of doubles
75
76     std::vector<int> ages;
77     ages.push_back(55);
78     ages.push_back(49);
79     ages.push_back(21);
80     ages.push_back(19);
81     params.set_param("Ages", ages);    // Vector of integers
82   }
83
84   {
85     // ============================================================
86     //+ EXAMPLE USAGE...
87     //+ Begin use of stk io heartbeat file...
88     stk::io::StkMeshIoBroker stkIo(communicator);
89
90     //+ Define the heartbeat output to be in TEXT format.
91     size_t hb = stkIo.add_heartbeat_output(file_name, stk::io::TEXT);
92
93     stk::util::ParameterMapType::const_iterator i = params.begin();
94     stk::util::ParameterMapType::const_iterator iend = params.end();
95     for (; i != iend; ++i) {
96       const std::string paramName = (*i).first;
97       //+ NOTE: A reference to the param is needed here.
98       stk::util::Parameter &param = params.get_param(paramName);
99
100      //+ Tell heartbeat which variables to output at each step...
101      //+ NOTE: The address of the value to be output is needed since the
102      //+       value is output in the process_heartbeat_output call.
103      stkIo.add_heartbeat_global(hb,paramName, param);
104    }
105
106    // Application's "Execution Loop"
107    int timestep_count = 1;
108    double time = 0.0;
109    for (int step=1; step <= timestep_count; step++) {
110      //+ Now output the global variables...
```

```
      //+ NOTE: All registered global values automatically output.
      stkIo.process_heartbeat_output(hb, step, time);
    }
  }
```

If the `stk::io::TEXT` argument to the `add_heartbeat_output()` function is changed to `stk::io::BINARY`, then the code will output a binary "history" file instead of a text-based file. Similarly for the other formats described above.

#### 5.1.32.1. Change output precision

The default precision of the floating point values written by heartbeat to the non-binary formats is five which gives a number of the form "-1.12345e+00". To change the precision, the application defines the "PRECISION" property prior to creating the heartbeat output. The lines below show how this is done and also select the `CSV` format. These lines would replace line 91 in the previous example.

**Listing 5.39 Writing global variables to a Heartbeat file in CSV format with extended precision**
**code/stk/stk_doc_tests/stk_io/usingHeartbeatCSVChangePrecision.cpp**

```
    //+ Output should have 10 digits of precision (1.0123456789e+00)
    //+             default precision is 5 digits (1.012345e+00)
    Ioss::PropertyManager hb_props;
    hb_props.add(Ioss::Property("PRECISION", 10));

    //+ Define the heartbeat output and the format (CSV)
    size_t hb = stkIo.add_heartbeat_output(file_name, stk::io::CSV, hb_props);

```

#### 5.1.32.2. Change field separator

Other customizations of the output are also possible. The example below shows the lines that would be changed in order to use a vertical bar "|" as the field separator in the `TEXT` format.

**Listing 5.40 Writing global variables to a Heartbeat file with a user-specified field separator**
**code/stk/stk_doc_tests/stk_io/usingHeartbeatOverrideSeparator.cpp**

```
    //+ Use vertical bar as field separator
    Ioss::PropertyManager hb_props;
    hb_props.add(Ioss::Property("FIELD_SEPARATOR", " | "));
    size_t hb =
        stkIo.add_heartbeat_output(file_name, stk::io::TEXT, hb_props);

```

### *5.1.33. Miscellaneous capabilities*

This section describes how to perform some functions that are useful, but don't fit into any of the previous sections.

#### 5.1.33.1. Add contents of a file and/or strings to the information records of a database

The first example shows how to embed the contents of a file into the information records of a results or restart output database. This is done on line 93. This is often useful since it then provides some documentation internal to the database itself showing the commands that were given to the application that created the database. The example also shows (see line 97) how to add a string as an additional information record.

In a parallel run in which the file-per-processor output is being used, the information records are only written to the file on processor 0.

**Listing 5.41 Adding the contents of a file to the information records of an output database**
**code/stk/stk_doc_tests/stk_io/addFileContentsToOutputDatabase.cpp**

```
61  // ============================================================
62  //+ SETUP
63  std::string input_file = "application_input_file.i";
64  std::string info1("This is the first line of the input file.");
65  std::string info2("This is the second line of the input file. "
66                    "It is longer than 80 characters, so it should be wrapped.");
67  std::string info3("This is the third line of the input file.");
68  std::string info4("This is the fourth and last line of the input file.");
69
70  std::string additional_info_record = "This is an info record added explicitly,"
71                                       " not from the input file.";
72  {
73    std::ofstream my_file(input_file.c_str());
74    my_file << info1 <<"\n" << info2 <<"\n" << info3 <<"\n" << info4 <<"\n";
75  }
76
77  {
78    // ============================================================
79    //+ EXAMPLE
80    stk::io::StkMeshIoBroker stkIo(communicator);
81    size_t ifh = stkIo.add_mesh_database("9x9x9|shell:xyzXYZ", "generated",
            stk::io::READ_MESH);
82    stkIo.set_active_mesh(ifh);
83    stkIo.create_input_mesh();
84    stkIo.populate_bulk_data();
85
86    // Output...
87    size_t fh = stkIo.create_output_mesh(filename,
88                                         stk::io::WRITE_RESULTS);
89    Ioss::Region *io_reg = stkIo.get_output_ioss_region(fh).get();
90
91    //+ Add the data from the file "application_input_file.i"
92    //+    as information records on this file.
93    io_reg->property_add(Ioss::Property("input_file_name",input_file));
94
95    //+ Add the data from the "additional_info_record" vector as
96    //+    information records on this file.
97    io_reg->add_information_record(additional_info_record);
98
99    stkIo.write_output_mesh(fh);
100   // ... Verification deleted
101
```

#### 5.1.33.2. Tell database to overwrite steps instead of adding new steps

The next example shows how to tell an output database (typically restart) to only store a single time step and overwrite this time step each time that a new step is added to the database. This is done by setting the cycle count on the database to one as is shown on line 83. The reason an application would want to do this is to minimize the size of a restart file, but still output restart data periodically in case the analysis job crashes for some reason.

For more robustness, an application might have two or more restart databases active and cycle writing to each database in turn. That is, if the application had two restart databases and it was writing every 0.1 seconds, it would write to the first database at times 0.1, 0.3, 0.5, 0.7; and it would write to the second database at times 0.2, 0.4. 0.6, 0.8. In this scenario, a crash during the output of one database would not affect the other database, so there should always be a database containing valid data.

**Listing 5.42 Overwriting time steps instead of adding new steps to a database**
**code/stk/stk_doc_tests/stk_io/singleStepOnRestart.cpp**

```
72       // ... Setup deleted
73       // ============================================================
74       // EXAMPLE USAGE...
75       // Create a restart file,
76       size_t fh = stkIo.create_output_mesh(filename,
77                                             stk::io::WRITE_RESTART);
78       stkIo.add_field(fh, field);
79
80       //+ Set the cycle count to 1.  This will result in a maximum
81       //+ of one step on the output database -- when a new step is
82       //+ added, it will overwrite the existing step.
83       stkIo.get_output_ioss_region(fh)->get_database()->set_cycle_count(1);
84
85       // Write multiple steps to the restart file.
86       for (size_t step=0; step < 3; step++) {
87         double time = step;
88         stkIo.begin_output_step(fh, time);
89         stkIo.write_defined_output_fields(fh);
90         stkIo.end_output_step(fh);
91       }
92
93       //+ At this point, there should only be a single state on the
94       //+ restart database. The time of this state should be 2.0.
95       // ... Verification deleted
96
```

The cycle count can be set to any value. In general, if the “analysis” step is “AS” and the cycle count is “CYCLE”, then the database step is given by “AS mod CYCLE” where “mod” is the remainder when AS is divided by CYCLE.

### 5.1.34. How to create and write a nodeset and sideset with fields using STK Mesh

**Listing 5.43 Example of creating and writing a nodeset with fields.
code/stk/stk_doc_tests/stk_io/howToCreateAndWriteNodesetOrSideset.cpp**

```
TEST_F(MeshWithNodeset, createAndWriteNodesetWithField)
{
  if (stk::parallel_machine_size(get_comm()) == 1)
  {
    setup_empty_mesh(stk::mesh::BulkData::AUTO_AURA);
    std::string nodesetName("nodelist_1");
    stk::mesh::Part& nodesetPart = get_meta().declare_part(nodesetName,
             stk::topology::NODE_RANK);

    const std::string fieldName = "nodesetField";
    const unsigned fieldLength = 3;
    double initialValue[fieldLength] {0., 0., 0.};
    const int numStates = 1;
    stk::mesh::Field<double> &newField =
        get_meta().declare_field<double>(stk::topology::NODE_RANK, fieldName, numStates);

    stk::mesh::put_field_on_mesh(newField, nodesetPart, fieldLength, initialValue);
    stk::io::set_field_output_type(newField, stk::io::FieldOutputType::VECTOR_3D);

    stk::io::fill_mesh("generated:1x1x1", get_bulk());

    stk::mesh::Entity node1 = get_bulk().get_entity(stk::topology::NODE_RANK, 1);

    get_bulk().modification_begin();
    get_bulk().change_entity_parts(node1, stk::mesh::ConstPartVector{&nodesetPart});
    get_bulk().modification_end();

    stk::io::put_io_part_attribute(nodesetPart);

    verify_field_is_valid(get_meta(), node1, initialValue, fieldLength, fieldName);
    verify_nodesetField_in_file(get_bulk(), node1, nodesetName, fieldName);
  }
}
```

**Listing 5.44 Example of creating and writing a sideset with fields.
code/stk/stk_doc_tests/stk_io/howToCreateAndWriteNodesetOrSideset.cpp**

```
TEST_F(MeshWithSideset, createAndWriteSidesetWithField)
{
  if (stk::parallel_machine_size(get_comm()) == 1)
  {
    setup_empty_mesh(stk::mesh::BulkData::AUTO_AURA);
    std::string sidesetName("surface_1");
    stk::mesh::Part& sidesetPart = get_meta().declare_part(sidesetName,
             get_meta().side_rank());

    const std::string fieldName = "sidesetField";
    const unsigned fieldLength = 3;
    double initialValue[fieldLength] {1., 1., 1.};
    const int numStates = 1;
    stk::mesh::Field<double> &newField =
        get_meta().declare_field<double>(get_meta().side_rank(), fieldName, numStates);

    stk::mesh::put_field_on_mesh(newField, sidesetPart, fieldLength, initialValue);
    stk::io::set_field_output_type(newField, stk::io::FieldOutputType::VECTOR_3D);

    stk::io::fill_mesh("generated:1x1x1", get_bulk());

    stk::mesh::Entity elem1 = get_bulk().get_entity(stk::topology::ELEM_RANK, 1);
    unsigned sideOrdinal = 0;
```

```
    get_bulk().modification_begin();
    stk::mesh::Entity side = get_bulk().declare_element_side(elem1, sideOrdinal,
             stk::mesh::PartVector{&sidesetPart});
    get_bulk().modification_end();

    stk::io::put_io_part_attribute(sidesetPart);

    verify_field_is_valid(get_meta(), side, initialValue, fieldLength, fieldName);
    verify_sidesetField_in_file(get_bulk(), side, sidesetName, fieldName);
  }
}
```

### 5.1.35. Nodal Ordering for Mesh Output

For applications that depend on nodal ordering in the mesh output file, it may be useful to observe that STK automatically orders the list of nodes that are written to the file according to the global ID in ascending order in memory. This is in contrast to Framework output, which writes the nodes in bucket ordering without sorting the global IDs. This may cause unexpected changes in applications that expect the original ordering found in the input mesh file.

# 6. STK COUPLING

STK Coupling is a wrapper module to MPI routines that manage MPI communicators. This module provides simplified interfaces to MPI communicator splits and inter-communicator operations.

## 6.1. SplitComms

`SplitComms` class allows splitting a MPI communicator into subcommunicators based on provided *colors*. *Color* is a non-negative integer that is used to group MPI processes into split communicators. Processes with same *color* will be placed into the same communicator after split. Additionally, upon the construction of `SplitComms`, pairwise communicators between split communicators will be created internally to establish one-to-one communication pattern between groups of split communicators.

### *6.1.1. Example of SplitComms usage*

**Listing 6.1 SplitComms usage example**
**code/stk/stk_doc_tests/stk_coupling/BasicCommSplit.cpp**

```
TEST(StkCouplingDocTest, split_comms)
{
  auto commWorld = MPI_COMM_WORLD;
  auto rank = stk::parallel_machine_rank(commWorld);
  auto commSize = stk::parallel_machine_size(commWorld);

  if (commSize < 2) GTEST_SKIP();

  auto color = rank % 2;

  stk::coupling::SplitComms splitComms(commWorld, color);
  splitComms.set_free_comms_in_destructor(true);

  auto subComm = splitComms.get_split_comm();
  std::vector<int> otherColors = splitComms.get_other_colors();
  EXPECT_EQ(1u, otherColors.size());

  for (auto otherColor : otherColors) {
    auto otherComm = splitComms.get_pairwise_comm(otherColor);

    int result;
    MPI_Comm_compare(subComm, otherComm, &result);
    if (color != otherColor) {
      EXPECT_NE(MPI_IDENT, result);
    } else {
      EXPECT_EQ(MPI_IDENT, result);
    }

    EXPECT_EQ(splitComms.get_parent_comm(), commWorld);
```

```
  }
}
```

* The `SplitComms` API can be found in *stk/stk_coupling/stk_coupling/SplitComms.hpp*.

### 6.1.2. SplitCommsSingleton

*STK Coupling* provides a capability to register a `SplitComms` object as a singleton object, allowing it to be referenced uniformly within a translation unit.

#### 6.1.2.1. Example of SplitCommsSingleton usage

**Listing 6.2 SplitComms usage example**
**code/stk/stk_doc_tests/stk_coupling/BasicCommSplit.cpp**

```
TEST(StkCouplingDocTest, split_comms_singleton)
{
  auto commWorld = MPI_COMM_WORLD;
  auto rank = stk::parallel_machine_rank(commWorld);
  auto color = rank % 2;

  stk::coupling::SplitComms splitComms(commWorld, color);
  splitComms.set_free_comms_in_destructor(true);
  stk::coupling::set_split_comms_singleton(splitComms);

  auto singletonComms = stk::coupling::get_split_comms_singleton();
  EXPECT_TRUE(singletonComms.is_initialized());

  int result;
  MPI_Comm_compare(splitComms.get_split_comm(), singletonComms.get_split_comm(), &result);
  EXPECT_EQ(MPI_IDENT, result);
}
```

* The `SplitCommsSingleton` API can be found in *stk/stk_coupling/stk_coupling/SplitCommsSingleton.hpp*.

## 6.2. SyncInfo

`SyncInfo` class can be used to perform inter-communicators data exchange. Using `SplitComms`, `SyncInfo` identifies internally stored pairwise communicators to exchange data between them.

* The `SyncInfo` API can be found in *stk/stk_coupling/stk_coupling/SyncInfo.hpp*.

### 6.2.1. Data Exchange

To exchange data between processors in split communicators, `SyncInfo::exchange()` can be used. Two overloaded `exchange` functions are available.

```
SyncInfo exchange(const SplitComms & splitComms, int otherColor) const
```

This `exchange` function can be used to perform data exchange between two communicators known by provided `SplitComms` object. Using internally stored *pairwise communicators*, root process of the caller's communicator and root process of *otherColor's* communicator exchange their stored data. Broadcasts within respective communicators follows, ensuring that all processes are given a copy of exchanged data. This function returns a newly constructed `SyncInfo` that has access to received data from the other communicator.

It should be noted that communication between communicators are only done between root processes of communicators. Thus, data that are expected to be exchanged must be present in `SyncInfo` of root process.

#### 6.2.1.1. Example of exchange() with two colors

**Listing 6.3 SyncInfo exchange with two colors example**
**code/stk/stk_doc_tests/stk_coupling/BasicCommSplit.cpp**

```cpp
TEST(StkCouplingDocTest, sync_info_exchange_two_colors)
{
  using stk::coupling::SplitComms;
  using stk::coupling::SyncInfo;

  auto commWorld = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(commWorld) != 4) GTEST_SKIP();

  auto rank = stk::parallel_machine_rank(commWorld);
  auto color = rank % 2;

  SplitComms splitComms(commWorld, color);
  SyncInfo syncInfo("exchange_info");

  std::string stringValue("DataFrom" + std::to_string(color));
  std::vector<int> intVector = (color == 0) ? std::vector<int>{1, 3, 5} : std::vector<int>{2, 4, 6, 8};
  std::vector<std::pair<std::string, double>> color0_vectorPairStringDouble = {{"one", 1.0}, {"two", 2.0}};
  std::vector<std::pair<std::string, double>> color1_vectorPairStringDouble = {{"three", 3.0}};
  std::vector<std::pair<std::string, double>> vectorPairStringDouble =
      (color == 0) ? color0_vectorPairStringDouble : color1_vectorPairStringDouble;

  syncInfo.set_value("stringToExchange", stringValue);
  syncInfo.set_value("vectorOfIntToExchange", intVector);
  syncInfo.set_value("vectorOfPairToExchange", vectorPairStringDouble);

  auto otherColors = splitComms.get_other_colors();
  SyncInfo exchangeInfo = syncInfo.exchange(splitComms, otherColors[0]);

  std::string exepctedStringValue("DataFrom" + std::to_string(otherColors[0]));
  std::vector<int> expectedIntVector = (color == 1) ? std::vector<int>{1, 3, 5} : std::vector<int>{2, 4, 6, 8};
  std::vector<std::pair<std::string, double>> expectedVectorPairStringDouble =
      (color == 1) ? color0_vectorPairStringDouble : color1_vectorPairStringDouble;

  auto recvString = exchangeInfo.get_value<std::string>("stringToExchange");
  auto recvVectorOfInt = exchangeInfo.get_value<std::vector<int>>("vectorOfIntToExchange");
  auto recvVectorOfPair = exchangeInfo.get_value<std::vector<std::pair<std::string, double>>>("vectorOfPairToExchange");
```

```
  EXPECT_EQ(exepctedStringValue, recvString);
  EXPECT_EQ(expectedIntVector, recvVectorOfInt);
  EXPECT_EQ(expectedVectorPairStringDouble, recvVectorOfPair);
}
```

2 `ColorToSyncInfoMap exchange(const SplitComms & splitComms) const`

This `exchange` function is used to emulate n-way data exchange with all other communicators known by `SplitComms` object. After a round of data exchange with all other communicators, a `ColorToSyncInfoMap` that contains *{color, SyncInfo}* key-value pairs is created and returned.

#### 6.2.1.2. Example of exchange() with multiple colors

**Listing 6.4 SyncInfo exchange with multiple colors example**
**code/stk/stk_doc_tests/stk_coupling/BasicCommSplit.cpp**

```
TEST(StkCouplingDocTest, sync_info_exchange_multi_colors)
{
  using stk::coupling::SplitComms;
  using stk::coupling::SyncInfo;

  auto commWorld = MPI_COMM_WORLD;
  auto commSize = stk::parallel_machine_size(commWorld);

  if (commSize % 3 != 0) GTEST_SKIP();

  auto rank = stk::parallel_machine_rank(commWorld);
  auto color = rank;
  auto intToExchange = color * (commSize / 3);

  SyncInfo syncInfo("exchange_info");
  SplitComms splitComms(commWorld, color);

  syncInfo.set_value<int>("value", intToExchange);

  SyncInfo::ColorToSyncInfoMap otherInfos = syncInfo.exchange(splitComms);

  std::for_each(otherInfos.begin(), otherInfos.end(), [&](const auto &mapElement) {
    [[maybe_unused]] int otherColor = mapElement.first;
    SyncInfo otherSyncInfo = mapElement.second;

    EXPECT_NE(intToExchange, otherSyncInfo.get_value<int>("value"));
    EXPECT_TRUE(otherSyncInfo.has_value<int>("value"));
  });
}
```

## 6.3. Miscellaneous

### *6.3.1. SyncInfo value comparison using SyncMode*

Values stored in two `SyncInfos` can be compared using `choose_value()`.

SyncMode is used to decide the values between two SyncInfos. Refer to the Table 6-1 for output cases:

**Table 6-1. Sync outcome based on SyncModes**

| SyncInfo_**A**\SyncInfo_**B** | SEND | RECV | MINIMUM | ANY |
|---|---|---|---|---|
| SEND | Throws | Value from A | Compute min | Value from A |
| RECV | Value from B | Throws | Compute min | Value from B |
| MINIMUM | Compute min | Compute min | Compute min | Compute min |
| ANY | Value from B | Value from A | Compute min | Throws |

#### 6.3.1.1. Example of choose_value() usage

**Listing 6.5 choose_values definition**
**code/stk/stk_doc_tests/stk_coupling/BasicCommSplit.cpp**

```
TEST(StkCouplingDocTest, sync_info_choose_values)
{
  using stk::coupling::SplitComms;
  using stk::coupling::SyncInfo;
  using stk::coupling::SyncMode;

  SyncInfo syncInfo("sync_info");
  SyncInfo otherInfo("other_sync_info");
  const std::string parameterName = "time_step";

  syncInfo.set_value(parameterName, 1.0);
  otherInfo.set_value(parameterName, 2.0);

  EXPECT_DOUBLE_EQ(stk::coupling::choose_value(syncInfo, otherInfo, parameterName,
             SyncMode::Send), 1.0);
  EXPECT_DOUBLE_EQ(stk::coupling::choose_value(syncInfo, otherInfo, parameterName,
             SyncMode::Receive), 2.0);
  EXPECT_DOUBLE_EQ(stk::coupling::choose_value(syncInfo, otherInfo, parameterName,
             SyncMode::Minimum), 1.0);
  EXPECT_DOUBLE_EQ(stk::coupling::choose_value(syncInfo, otherInfo, parameterName,
             SyncMode::Any), 1.0);
}
```

### 6.3.2. *Reserved parameter names*

Following names are predefined in *STK Coupling* and are reserved.

**Listing 6.6 Reserved Names**
**code/stk/stk_coupling/stk_coupling/Constants.hpp**

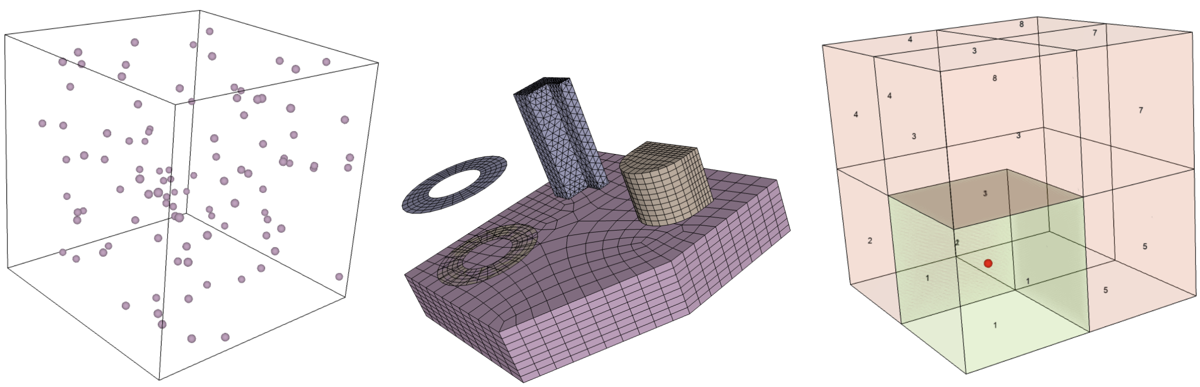
```
static const std::string AppName = "Application Name";
static const std::string TimeSyncMode = "Time Sync Mode";
static const std::string InitialTime = "Initial Time";
static const std::string CurrentTime = "Current Time";
static const std::string TimeStep = "Time Step";
static const std::string FinalTime = "Final Time";
static const std::string IsFinished = "Is Finished";
static const std::string SuccessFlag = "Is Successful";
```

### 6.3.3. *Version Compatibility*

Multiple executables that use the *STK Coupling* modules can be launched as a single MPMD MPI job. During the execution, the *STK Coupling* module checks for *STK Coupling* versions in all translation units as `SplitComms` object is initiated. If any mismatch between *STK Coupling* module versions is detected, the module will abort and information on *STK* module version incompatibility will be output.

# 7. STK SEARCH

The STK *Search* module provides a geometric proximity search capability that allows for the determination of relationships on a collection of geometric objects.



**(a)** Particle Mesh **(b)** Contact Mesh **(c)** Interpolation Mesh

**Figure 7-1. Geometric Search Usage Examples**

Proximity searches are an important component of various physics applications such as

- Nearest neighbor:
  Figure 7-1a shows an example of a domain with a distribution of particles where it is necessary to either compute the nearest neighbor or the closest set of particles within a given distance for collision calculations.
- Contact search:
  Figure 7-1b shows an example of a contact search calculation where disparate objects that are meshed dis-contiguously have to be treated as one object. This is a common technique in many fields of computational mechanics. Contact search is done between the external facets of the objects and enforcement constraints are set up in the system of equations to ensure continuity.
- Interpolation:
  Figure 7-1c shows an example where it is necessary to perform interpolation of a field defined on a mesh domain, unto a point that lies within this domain. A geometric search has

to be performed to determine the nearest enclosing element from which the interpolation may be done. This is an important technique used in various capabilities such as field data probes and in data transfer between two computational domains, where computed field values from a source mesh may be transferred and used as variables for physics on a destination mesh with a possible different discretization and/or length scale.

STK Search supports two broad "categories" of search, which we refer to as *Coarse Search* and *Fine Search*.

*Coarse Search* is a geometric proximity search which is performed using geometric objects such as axis-aligned bounding boxes, spheres, or points. This coarse search is not dependent on domain-specific data such as a mesh, etc., although it is common to construct the input boxes/spheres/points from mesh elements or nodes, etc.

*Fine Search* essentially consists of filtering the results of a coarse search to ensure precise correspondence to the related mesh objects. For example, if the coarse search is performed using axis- aligned bounding boxes constructed from unstructured-mesh elements, any point that was found to be inside a bounding box must be confirmed to actually be inside the mesh element (which is not necessarily axis-aligned). The fine search filtering typically requires an abstraction layer or "plugin" approach to incorporate user-specific capabilities such as finite-element shape functions and parametric coordinate evaluations etc. The *Fine Search* section includes considerable discussion of mesh "wrapper" infrastructure for incorporating the needed capabilities without requiring a specific mesh implementation such as STK Mesh.

## 7.1. Coarse Search

Coarse search is performed by intersection tests on bounding shapes which are defined using the collection of geometric entities. The overall methodology for coarse search is to group these geometric entities into *domain* and *range* entities on which proximity searches may be performed. The result of the coarse search identifies pairs of domain and range objects that were determined to intersect.

Figure 7-2 shows a single point that straddles the edge boundary of 4 elements in a mesh of 8 elements. In this case, the search should identify elements {1, 3, 5, 7} as containing the point.

Bounding box searches can be performed via a number of fast algorithms, many of which are tree based and operate with logarithmic complexity. Currently, STK Search implements a *KDTREE* algorithm for CPU/Host builds and a *MORTON_LBVH* (Morton Linear Bounding Volume Hierarchy) algorithm for use on CPU (host) or GPU (device). Additionally, an option to use the *ArborX* library is also provided. It should be noted that the *ArborX* library is often the fastest search option, especially for GPU problems (as compared to the *MORTON_LBVH* implementation), for many of the data sets that we have measured.

STK Search provides `coarse_search` and `local_coarse_search` functions. The `coarse_search` function is MPI parallel, and can find off-processor intersections. By default, results include any intersections between local domain objects and remote range objects. Optionally, (controlled by the flag `enforceSearchResultSymmetry`), a final

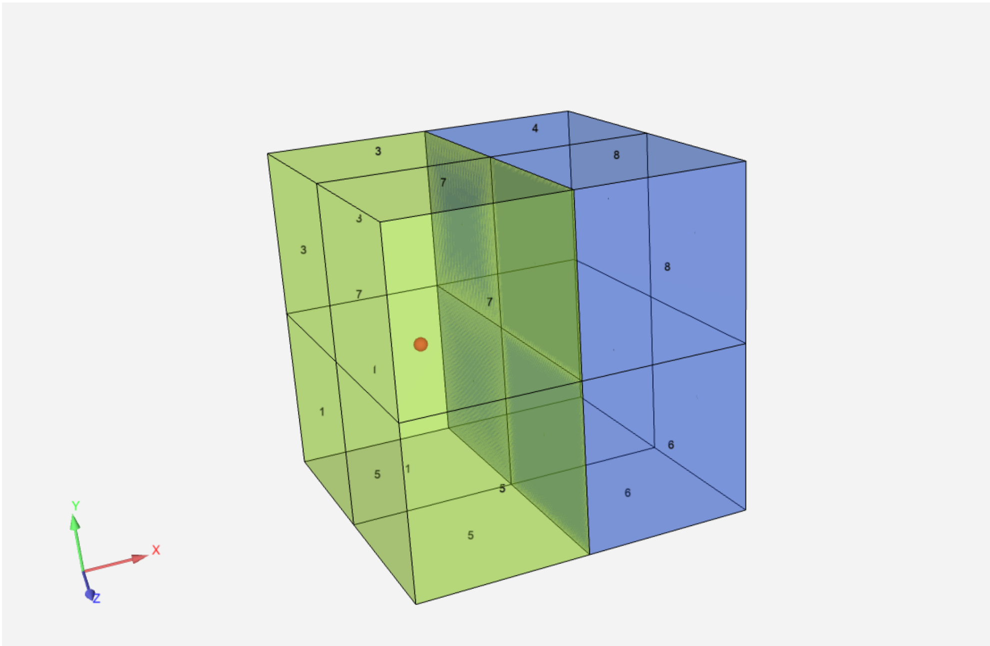



**Figure 7-2. Simple STK Search**

communication step is performed to include any intersections between local range objects and remote domain objects. The `local_coarse_search` function is purely local, and doesn't have any knowledge of MPI-parallel considerations.

The `coarse_search` and `local_coarse_search` functions each have overloads that accept inputs and return results in either `std::vector` objects or in `Kokkos::View` objects. The `std::vector` versions execute only on the CPU host, even if a GPU device is available. The `Kokkos::View` versions can execute on a GPU device or on the CPU host, depending on the *Kokkos execution-space* argument specified by the caller. A `SearchMethod` argument is used to specify the desired search algorithm. Valid options for search method are `KDTREE`, `ARBORX` and `MORTON_LBVH`. Note that the *ARBORX* option is only valid if the third-party-library *ArborX* is installed and enabled in the STK Search build.

Supported combinations of search methods and search implementations are shown in the following table. Note that the `std::vector` versions of `coarse_search` and `local_coarse_search` are denoted by *(Host)* while the `Kokkos::View` versions are denoted by *(Host/Device)*.

**Table 7-1. Search options**

| Search Method | Coarse Search (Host) | Coarse Search (Host/Device) | Local Coarse Search (Host) | Local Coarse Search (Host/Device) |
|---|---|---|---|---|
| ARBORX | ✓ | ✓ | ✓ | ✓ |
| KDTREE | ✓ | DISABLED | ✓ | DISABLED |
| MORTON_LBVH | ✓ | ✓ | ✓ | ✓ |

The `coarse_search` and `local_coarse_search` functions are templated in order to provide the flexibility of specifying inputs as objects that may be boxes, spheres or points. Additionally, each object is paired with an identifier for use in mapping results back to user-inputs.

For the MPI parallel search, the identifier is in turn a pair that includes the user-provided identifier and the processor-rank. Thus the items in the input `std::vector` objects are of type `std::pair<DomainBoxType,DomainIdentProcType>`. For the `Kokkos::View` objects, the contained items are `BoxIdentProc<DomainBoxType,DomainIdentProcType>`. This is best illustrated by an example, which can be seen in the documentation-test `stk_doc_tests/stk_search/howToNgpElemNodeNeighbors`. This example performns a search using a "domain" of spheres at element centers, and finds the node "neighbors" of each element by searching against a "range" of node points. This example program illustrates using the MPI parallel `Kokkos::View` version of `coarse_search`. The input data (domain and range spheres and points) is constructed on the GPU device, and the search results are returned in a device-resident `Kokkos::View`.

This first code snippet shows how the above-mentioned types are defined in the program.

**Listing 7.1 GPU Coarse Search definition of types**
**code/stk/stk_doc_tests/stk_search/howToNgpSearchElemNodeNeighbors.cpp**

```
using ElemIdentProc = stk::search::IdentProc<unsigned,int>;
using NodeIdentProc = stk::search::IdentProc<stk::mesh::EntityId,int>;
using SphereIdentProc = stk::search::BoxIdentProc<stk::search::Sphere<double>,ElemIdentProc>;
using PointIdentProc = stk::search::BoxIdentProc<stk::search::Point<double>,NodeIdentProc>;
using Intersection = stk::search::IdentProcIntersection<ElemIdentProc,NodeIdentProc>;

using DomainViewType = Kokkos::View<SphereIdentProc*,ExecSpace>;
using RangeViewType = Kokkos::View<PointIdentProc*,ExecSpace>;
using ResultViewType = Kokkos::View<Intersection*,ExecSpace>;
```

This example program illustrates the usage of `coarse_search` using data constructed from an instance of STK Mesh for convenience, although it is worth emphasizing again that STK Search doesn't depend on or require STK Mesh.

This next code snippet shows the creation of the node-point objects on GPU.

**Listing 7.2 GPU Coarse Search construction of node points**
**code/stk/stk_doc_tests/stk_search/howToNgpSearchElemNodeNeighbors.cpp**

```
  Kokkos::parallel_for(stk::ngp::DeviceRangePolicy(0, numLocalNodes),
    KOKKOS_LAMBDA(const unsigned& i) {
      stk::mesh::EntityFieldData<double> coords = ngpCoords(nodeIndices(i));
      stk::mesh::Entity node = ngpMesh.get_entity(stk::topology::NODE_RANK, nodeIndices(i));
      nodePoints(i) = PointIdentProc{stk::search::Point<double>(coords[0], coords[1],
             coords[2]), NodeIdentProc(ngpMesh.identifier(node), myRank)};
    }
  );
```

The construction of the element-spheres is not shown here but it may be seen in the source code. This next code snippet shows the call to the `coarse_search` method.

**Listing 7.3 GPU Coarse Search usage example**
**code/stk/stk_doc_tests/stk_search/howToNgpSearchElemNodeNeighbors.cpp**

```
  DomainViewType elemSpheres = create_elem_spheres(*meshPtr, radius);
  RangeViewType nodePoints = create_node_points(*meshPtr);

  EXPECT_EQ(elemSpheres.size(), numLocalElems);
```

```
EXPECT_EQ(nodePoints.size(), numLocalOwnedNodes);

ResultViewType searchResults;
stk::search::SearchMethod searchMethod = stk::search::MORTON_LBVH;

stk::ngp::ExecSpace execSpace = Kokkos::DefaultExecutionSpace{};
const bool enforceSearchResultSymmetry = true;
MPI_Comm comm = meshPtr->parallel();

stk::search::coarse_search(elemSpheres, nodePoints, searchMethod, comm, searchResults,
           execSpace, enforceSearchResultSymmetry);
```

Most of the example program is not shown here, but all of the details may be found in the source code. The program goes on to import mesh data corresponding to remote search intersections, and unpacks the search results into a mesh field.

## 7.2. Fine Search

Fine search is a post-processing stage to coarse search in which filtering for a destination mesh entity is performed on the list of candidate source entities in order to select the best candidate. Two primary metrics are used to determine what is considered the best candidate and these are *parametric* and *geometric* distances.

### *7.2.1. Parametric distance metric*

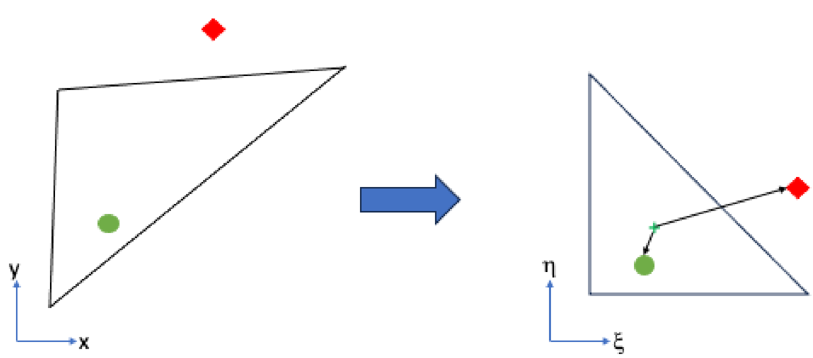



**Figure 7-3. Parametric distance calculation**

Figure 7-3 shows a planar element which has been mapped from physical space (x,y) to parametric space $(\varepsilon, \eta)$. Based on the *parametric coordinates* for a physical point, inside or outside the element, a parametric distance can be computed in this parametric space and used as a metric to select a best candidate from a list of candidates.

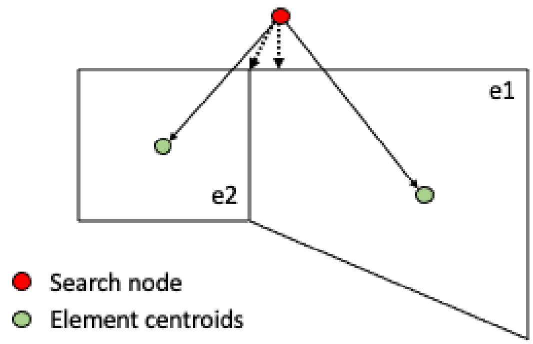



**Figure 7-4. Geometric distance calculation**

### 7.2.2. *Geometric distance metric*

Figure 7-4 shows an alternative to the filtering algorithm for determining the best entity from a list of candidates through geometric considerations. Two options are displayed which are the distance from the search point to the element centroid as well as a geometric projection from the point to the boundary element. The boundary projection is especially useful for situations where the search point is outside the candidate element.

## 7.3. Mesh Interface

Extendable interfaces to source and destination mesh are provided as guidelines for defining *STK Search* meshes. Interfaces can be extended to perform compile-time check on expected user-defined functions and template specializations. Alternatively, custom implementations of source and destination meshes can be used without extending provided interfaces. However, all required functions from *Coarse Search* and *Fine Search* interfaces are expected to be fully defined.

**Listing 7.4 StkSearch interface**
**code/stk/stk_search/stk_search/SearchInterface.hpp**

```
template <typename MESH>
struct MeshTraits;

template <typename SENDMESH>
class SourceMeshInterface
{
 public:
  using Entity = typename MeshTraits<SENDMESH>::Entity;
  using EntityVec = typename MeshTraits<SENDMESH>::EntityVec;
  using EntityKey = typename MeshTraits<SENDMESH>::EntityKey;
  using EntityProc = typename MeshTraits<SENDMESH>::EntityProc;
  using EntityProcVec = typename MeshTraits<SENDMESH>::EntityProcVec;
  using Point = typename MeshTraits<SENDMESH>::Point;
  using Box = typename MeshTraits<SENDMESH>::Box;
  using BoundingBox = typename MeshTraits<SENDMESH>::BoundingBox;

  SourceMeshInterface() = default;
  virtual ~SourceMeshInterface() = default;
```

```
  virtual stk::ParallelMachine comm() const = 0;

  virtual std::string name() const = 0;

  virtual void bounding_boxes(std::vector<BoundingBox>& boxes) const = 0;

  virtual void find_parametric_coords(
    const EntityKey k,
    const double* toCoords,
    std::vector<double>& parametricCoords,
    double& parametricDistance,
    bool& isWithinParametricTolerance) const = 0;

  virtual bool modify_search_outside_parametric_tolerance(
    const EntityKey k,
    const double* toCoords,
    std::vector<double>& parametricCoords,
    double& geometricDistanceSquared,
    bool& isWithinGeometricTolerance) const = 0;

  virtual double get_distance_from_nearest_node(
    const EntityKey k, const double* point) const = 0;

  virtual double get_closest_geometric_distance_squared(
    const EntityKey k, const double* toCoords) const = 0;

  virtual double get_distance_from_centroid(
    const EntityKey k, const double* toCoords) const = 0;

  virtual double get_distance_squared_from_centroid(
    const EntityKey k, const double* toCoords) const = 0;

  virtual void centroid(const EntityKey k, std::vector<double>& centroidVec) const = 0;

  virtual const double* coord(const EntityKey k) const = 0;
};

template <typename RECVMESH>
class DestinationMeshInterface
{
 public:
  using Entity = typename MeshTraits<RECVMESH>::Entity;
  using EntityVec = typename MeshTraits<RECVMESH>::EntityVec;
  using EntityKey = typename MeshTraits<RECVMESH>::EntityKey;
  using EntityProc = typename MeshTraits<RECVMESH>::EntityProc;
  using EntityProcVec = typename MeshTraits<RECVMESH>::EntityProcVec;
  using Point = typename MeshTraits<RECVMESH>::Point;
  using Sphere = typename MeshTraits<RECVMESH>::Sphere;
  using BoundingBox = typename MeshTraits<RECVMESH>::BoundingBox;

  DestinationMeshInterface() = default;
  virtual ~DestinationMeshInterface() = default;

  virtual stk::ParallelMachine comm() const = 0;

  virtual std::string name() const = 0;

  virtual void bounding_boxes(std::vector<BoundingBox>& v) const = 0;

  virtual const double* coord(const EntityKey k) const = 0;
  virtual double get_search_tolerance() const = 0;
  virtual double get_parametric_tolerance() const = 0;

  virtual void centroid(const EntityKey k, std::vector<double>& centroidVec) const = 0;
  virtual double get_distance_from_nearest_node(
    const EntityKey k, const double* toCoords) const = 0;
};
```

#### 7.3.1. Source mesh

- `stk::ParallelMachine comm() const` : returns a `stk::ParallelMachine` variable, which is a typedef for `MPI_Comm` in *STK*. [coarse search] [fine search]
- `std::string name() const` : returns the user-defined name of the mesh. [coarse search] [fine search]
- `void bounding_boxes(std::vector<T>&) const` : populates input vector with bounding box information that will be inserted into kdtree-based coarse search. [coarse search]

**Listing 7.5 Bounding boxes example**
**code/stk/stk_doc_tests/stk_search/searchMockMesh.hpp**

```
void bounding_boxes(std::vector<BoundingBox>& v) const
{
  Point center(m_coords[0], m_coords[1], m_coords[2]);

  EntityKey key = 1;
  EntityProc theIdent(key, stk::parallel_machine_rank(m_comm));
  BoundingBox theBox(Sphere(center, m_geometricTolerance), theIdent);
  v.push_back(theBox);
}

```

- `bool modify_search_outside_parametric_tolerance(const EntityKey, const double*, std::vector<double>&, double&, bool&) const` : determines search behavior if coordiante is outside of domain. If the function returns true, then input variables are expected to be modified. [fine search]
- `double get_distance_from_nearest_node(const EntityKey, const double*) const` : returns geometric distance between input coordinate and the nearest node of input entity. [fine search]

**Listing 7.6 get_distance_from_nearest_node example**
**code/stk/stk_doc_tests/stk_search/searchMockMesh.hpp**

```
double get_distance_from_nearest_node(const EntityKey k, const double* point) const
{
  const stk::mesh::Entity e = m_bulk.get_entity(k);

  STK_ThrowRequireMsg(
      m_bulk.entity_rank(e) == stk::topology::ELEM_RANK, "Invalid entity rank for
           object: " << m_bulk.entity_rank(e));

  double minDistance = std::numeric_limits<double>::max();
  const unsigned nDim = m_meta.spatial_dimension();

  const stk::mesh::Entity* const nodes = m_bulk.begin_nodes(e);
  const int num_nodes = m_bulk.num_nodes(e);

  for (int i = 0; i < num_nodes; ++i) {
    double d = 0.0;
```

```
      double* node_coordinates =
            static_cast<double*>(stk::mesh::field_data(*m_coordinateField, nodes[i]));

      for (unsigned j = 0; j < nDim; ++j) {
        const double t = point[j] - node_coordinates[j];
        d += t * t;
      }
      if (d < minDistance) minDistance = d;
    }

    minDistance = std::sqrt(minDistance);
    return minDistance;
  }

```

- `double get_closest_geometric_distance_squared(const EntityKey, const double* toCoords) const` : returns geometric distance squared from the nearest node. [fine search]
- `double get_distance_from_centroid(const Entitykey, const double*) const` : returns geometric distance from input entity's centroid. [fine search]
- `double get_distance_squared_from_centroid(const EntityKey, const double*) const` : returns geometric distance from input entity's centroid. [fine search]
- `void centroid(const EntityKey, std::vector<double>&) const` : populates input vector with centroid information of input entity. [fine search]
- `const double* coord(const EntityKey) const` : returns the coordinate of input entity. [fine search]

### *7.3.2. Destination Mesh*

- `stk::ParallelMachine comm() const` : returns a `stk::ParallelMachine` variable, which is a typedef for `MPI_Comm` in *STK*. [coarse search] [fine search]
- `std::string name() const` : returns the user-defined name of the mesh. [coarse search] [fine search]
- `void bounding_boxes(std::vector<T>&) const` : populates input vector with bounding box information that will be inserted into kdtree-based coarse search. [coarse search]

**Listing 7.7 Bounding boxes example**
**code/stk/stk_doc_tests/stk_search/searchMockMesh.hpp**

```
void bounding_boxes(std::vector<BoundingBox>& v) const
{
  Point center(m_coords[0], m_coords[1], m_coords[2]);

  EntityKey key = 1;
  EntityProc theIdent(key, stk::parallel_machine_rank(m_comm));
  BoundingBox theBox(Sphere(center, m_geometricTolerance), theIdent);
  v.push_back(theBox);
}

```

- `const double* coord(const EntityKey) const` : returns coordinate of input entity. [fine search]
- `double get_search_tolerance() const` : returns geometric tolerance for destination mesh. [fine search]
- `double get_parametric_tolerance() const` : return parametric tolerance for destination mesh. [fine search]
- `void centroid(const EntityKey, std::vector<double>&) const` : populates input vector with centroid information of input entity. [fine search]
- `double get_distance_from_nearest_node(const EntityKey, const double*) const` : returns geometric distance between input coordinate and the nearest node of input entity. [fine search]

An explicit template specialization of *MeshTrait* is required to be defined for both meshes by the user. *MeshTrait* is templated on mesh type and must include the type definition of *BoundingBox* and *EntityKey* for the mesh.

**Listing 7.8 MeshTrait example**
**code/stk/stk_doc_tests/stk_search/searchMockMesh.hpp**

```
template <>
struct MeshTraits<doc_test::SinglePointMesh> {
  using Entity = int;
  using EntityVec = std::vector<Entity>;
  using EntityKey = int;
  using EntityProc = stk::search::IdentProc<EntityKey, unsigned>;
  using EntityProcVec = std::vector<EntityProc>;
  using Point = stk::search::Point<double>;
  using Sphere = stk::search::Sphere<double>;
  using BoundingBox = std::pair<Sphere, EntityProc>;
};
```

### 7.3.3. *Fine Search*

#### 7.3.3.1. Fine Search API

**Listing 7.9 Filter Coarse Search**
**code/stk/stk_search/stk_search/FilterCoarseSearch.hpp**

```
template <class SENDMESH, class RECVMESH>
void filter_coarse_search(const std::string& name,
                          FilterCoarseSearchProcRelationVec<SENDMESH, RECVMESH>&
             rangeToDomain,
                          SENDMESH& sendMesh, RECVMESH& recvMesh,
                          FilterCoarseSearchOptions& filterOptions,
                          FilterCoarseSearchResult<RECVMESH>& filterResult)
```

- `const std::string&` : A string that will be used in the logistic output summary

- `FilterCoarseSearchProcRelationVec<SENDMESH, RECVMESH>&` : One dimensional vector of mappings between recv entities and candidate send entities
- `SENDMESH&` : source mesh
- `RECVMESH&` : destination mesh
- `FilterCoarseSearchOptions&` : A struct that contains options for controlling the algorithmic behavior of `filter_to_coarse_search`
- `FilterCoarseSearchResult<RECVMESH>&` : A result output data structure that will be populated from `filter_to_coarse_search`. This represents an interface; users are expected to extend and provide own implementation.

#### 7.3.3.2. Fine Search API Arguments

**Listing 7.10 Filter Coarse Search Options**
**code/stk/stk_search/stk_search/FilterCoarseSearch.hpp**

```
struct FilterCoarseSearchOptions
{
  std::ostream& m_outputStream{std::cout};
  ObjectOutsideDomainPolicy m_extrapolatePolicy{ObjectOutsideDomainPolicy::EXTRAPOLATE};
  bool m_useNearestNodeForClosestBoundingBox{false};
  bool m_useCentroidForGeometricProximity{false};
  bool m_verbose{true};
```

- `bool m_useNearestNodeForClosestBoundingBox` : Forces algorithm to be purely geometric by only considering the distance between search point and the nearest node on the candidate entity
- `bool m_useCentroidForGeometricProxmity` : If parametric distance check fails, then the algorithm switches to geometric distance check. If this option is set to true, geometric distance is computed using distance to the centroid of the candidate element. Otherwise, it is calculated using projection to the candidate entity.

**Listing 7.11 Object Outside Domain Policy**
**code/stk/stk_search/stk_search/FilterCoarseSearch.hpp**

```
enum class ObjectOutsideDomainPolicy { IGNORE, EXTRAPOLATE, TRUNCATE, PROJECT, ABORT,
               UNDEFINED_OBJFLAG = 0xff };
```

- `IGNORE` : Ignores the candidate entity if outside of the entity
- `EXTRAPOLATE` : If a search object lies outside of domain, the search result parametric coordinates are not modified
- `TRUNCATE` : If a search object lies outside of domain, the search result parametric coordinates are truncated to the boundary of the candidate element in parametric space
- `PROJECT` : If a search object lies outside of domain, the search result parametric coordinates are projected to the boundary of the candidate element in parametric space
- `ABORT` : Terminates search if any destination point lies outside of send domain

The following is the abstract interface for `FilterCoarseSearchResult`. Users are expected to extend this class and provide definition of abstract functions.

**Listing 7.12 Filter Coarse Search Result**
**code/stk/stk_search/stk_search/FilterCoarseSearch.hpp**

```
template <class RECVMESH>
class FilterCoarseSearchResult
{
public:
  using EntityKey = typename RECVMESH::EntityKey;

  virtual void add_search_filter_info(const EntityKey key,
                                      const std::vector<double>&paramCoords,
                                      const double parametricDistance,
                                      const bool isWithinParametricTolerance,
                                      const double geometricDistanceSquared,
                                      const bool isWithinGeometricTolerance) = 0;

  virtual void get_parametric_coordinates(const EntityKey key, std::vector<double>&
            paramCoords) const = 0;
  virtual void clear() = 0;

  virtual ~FilterCoarseSearchResult() {}
};
```

Two predefined derived classes of `FilterCoarseSearchResult` are provided using a `std::map` and a `std::vector`.

**Listing 7.13 Filter Coarse Search Result Map**
**code/stk/stk_search/stk_search/FilterCoarseSearch.hpp**

```
template <class RECVMESH>
class FilterCoarseSearchResultMap : public FilterCoarseSearchResult<RECVMESH>
```

**Listing 7.14 Filter Coarse Search Result Vector**
**code/stk/stk_search/stk_search/FilterCoarseSearch.hpp**

```
template <class RECVMESH>
class FilterCoarseSearchResultVector : public FilterCoarseSearchResult<RECVMESH>
```

## 7.4. STK Search Mesh Interface examples

Following is a sample implementation of *Coarse Search* and *Fine Search* usages:

### *7.4.1. Coarse Search example*

**Listing 7.15 Coarse Search wrapper example**
**code/stk/stk_doc_tests/stk_search/howToUseCoarseSearch.cpp**

```
  using CoarseSearchType = CoarseSearchTrait<Hex8SourceMesh, SinglePointMesh>;
  using Relation = typename CoarseSearchType::EntityProcRelation;
  using RelationVec = typename CoarseSearchType::EntityProcRelationVec;

```

```
  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) != 1) {
    GTEST_SKIP();
  }

  // Build 8 element cube
  const std::string meshSpec("generated:2x2x2");
  const unsigned spatialDim = 3;

  stk::mesh::MeshBuilder builder(communicator);
  builder.set_spatial_dimension(spatialDim);
  std::shared_ptr<stk::mesh::BulkData> mesh = builder.create();
  stk::mesh::MetaData& meta = mesh->mesh_meta_data();
  stk::io::fill_mesh(meshSpec, *mesh);

  // Point in element 1
  double x = 0.5, y = 1, z = 1;
  double geometricTolerance = 0.1;
  double parametricTolerance = 0.001;
  stk::mesh::EntityKey expectedSendKey(stk::topology::ELEM_RANK, 1u);

  // Create recv mesh
  auto recvMesh = std::make_shared<SinglePointMesh>(communicator, x, y, z,
             parametricTolerance, geometricTolerance);

  // Create send mesh
  stk::mesh::Part* part = meta.get_part("block_1");
  STK_ThrowRequireMsg(nullptr != part, "Error: block_1 does not exist");
  stk::mesh::PartVector parts{part};
  auto sendMesh = std::make_shared<Hex8SourceMesh>(*mesh, parts, mesh->parallel(),
             parametricTolerance);

  RelationVec coarseSearchResult;

  // Get single recv point
  SinglePointMesh::EntityKey expectedRecvKey(1);
  SinglePointMesh::EntityProc rangeEntry(expectedRecvKey, 0);

  double expansionFactor = 0.01;
  double expansionSum = 0.005;

  do_coarse_search<CoarseSearchType>(*sendMesh, *recvMesh, expansionFactor, expansionSum,
             coarseSearchResult);

  EXPECT_EQ(4u, coarseSearchResult.size());
```

**Listing 7.16 Coarse Search usage example**
**code/stk/stk_doc_tests/stk_search/howToUseCoarseSearch.cpp**

```
template <typename CoarseSearchType>
void do_coarse_search(typename CoarseSearchType::SendMesh& sendMesh,
    typename CoarseSearchType::RecvMesh& recvMesh,
    const double expansionFactor,
    const double expansionSum,
    typename CoarseSearchType::EntityProcRelationVec& coarseSearchResult)
{
  using SendBoundingBox = typename CoarseSearchType::SendBoundingBox;
  using RecvBoundingBox = typename CoarseSearchType::RecvBoundingBox;

  std::vector<SendBoundingBox> domain_vector;
  std::vector<RecvBoundingBox> range_vector;

  sendMesh.bounding_boxes(domain_vector);
  recvMesh.bounding_boxes(range_vector);

  if (!local_is_sorted(domain_vector.begin(), domain_vector.end(),
```

```
            BoundingBoxCompare<SendBoundingBox>()))
    std::sort(domain_vector.begin(), domain_vector.end(),
            BoundingBoxCompare<SendBoundingBox>());

  if (!local_is_sorted(range_vector.begin(), range_vector.end(),
            BoundingBoxCompare<RecvBoundingBox>()))
    std::sort(range_vector.begin(), range_vector.end(),
            BoundingBoxCompare<RecvBoundingBox>());

  for (SendBoundingBox& i : domain_vector) {
    inflate_bounding_box(i.first, expansionFactor, expansionSum);
  }

  stk::search::coarse_search(range_vector, domain_vector, stk::search::KDTREE,
            sendMesh.comm(), coarseSearchResult);

  std::sort(coarseSearchResult.begin(), coarseSearchResult.end());
}
```

#### 7.4.1.1. Coarse Search example

### *7.4.2. Fine Search example*

**Listing 7.17 Fine Search usage example**
**code/stk/stk_doc_tests/stk_search/howToUseFilterCoarseSearch.cpp**

```
TEST(StkSearchHowTo, useFilterCoarseSearch)
{
  using Relation = std::pair<SinglePointMesh::EntityProc, Hex8SourceMesh::EntityProc>;
  using RelationVec = std::vector<Relation>;

  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) != 1) { GTEST_SKIP(); }

  // Build 8 element cube
  const std::string meshSpec("generated:2x2x2");
  const unsigned spatialDim = 3;

  stk::mesh::MeshBuilder builder(communicator);
  builder.set_spatial_dimension(spatialDim);
  std::shared_ptr<stk::mesh::BulkData> mesh = builder.create();
  stk::mesh::MetaData& meta = mesh->mesh_meta_data();
  stk::io::fill_mesh(meshSpec, *mesh);

  // Point in element 1
  double x = 0.5, y = 0.5, z = 0.5;
  double geometricTolerance = 0.1;
  double parametricTolerance = 0.001;
  stk::mesh::EntityKey expectedSendKey(stk::topology::ELEM_RANK, 1u);

  // Create recv mesh
  auto recvMesh = std::make_shared<SinglePointMesh>(communicator, x, y, z,
            parametricTolerance, geometricTolerance);

  // Create send mesh
  stk::mesh::Part* part = meta.get_part("block_1");
  STK_ThrowRequireMsg(nullptr != part, "Error: block_1 does not exist");
  stk::mesh::PartVector parts{part};
  auto sendMesh = std::make_shared<Hex8SourceMesh>(*mesh, parts, mesh->parallel(),
            parametricTolerance);

  RelationVec relationVec;

  // Get single recv point
```

```
  SinglePointMesh::EntityKey expectedRecvKey(1);
  SinglePointMesh::EntityProc rangeEntry(expectedRecvKey, 0);

  // Load all elements as coarse search candidates
  stk::mesh::BucketVector const& buckets = mesh->get_buckets(stk::topology::ELEM_RANK,
             meta.universal_part());
  for(auto&& ib : buckets) {
    stk::mesh::Bucket& b = *ib;

    for(auto elem : b) {
      stk::mesh::EntityKey domainKey = mesh->entity_key(elem);
      Hex8SourceMesh::EntityProc domainEntry(domainKey, 0);

      relationVec.emplace_back(rangeEntry, domainEntry);
    }
  }

  EXPECT_EQ(8u, relationVec.size());

  bool useNearestNodeForClosestBoundingBox{false};
  bool useCentroidForGeometricProximity{false};
  bool verbose{false};
  auto extrapolateOption = stk::search::ObjectOutsideDomainPolicy::ABORT;

  stk::search::FilterCoarseSearchOptions options(std::cout, extrapolateOption,
                                                useNearestNodeForClosestBoundingBox,
                                                useCentroidForGeometricProximity, verbose);
  stk::search::FilterCoarseSearchResultVector<SinglePointMesh> searchResults;
  stk::search::filter_coarse_search("filter", relationVec, *sendMesh, *recvMesh, options,
             searchResults);

  EXPECT_EQ(1u, relationVec.size());

  auto relation = relationVec[0];
  const SinglePointMesh::EntityKey recvEntityKey = relation.first.id();
  const Hex8SourceMesh::EntityKey sendEntityKey = relation.second.id();

  EXPECT_EQ(expectedRecvKey, recvEntityKey);
  EXPECT_EQ(expectedSendKey, sendEntityKey);
}
```

This page intentionally left blank.

# 8. STK TRANSFER

STK Transfer provides an interface for transferring field data between meshes. `TransferBase` is a base class that defines the user-level API for using STK Transfer. Figure 8-1 shows the three primary derived classes in the STK Transfer library. Each of these three classes provides a unique transfer capability, which will be described in detail in the following sections.

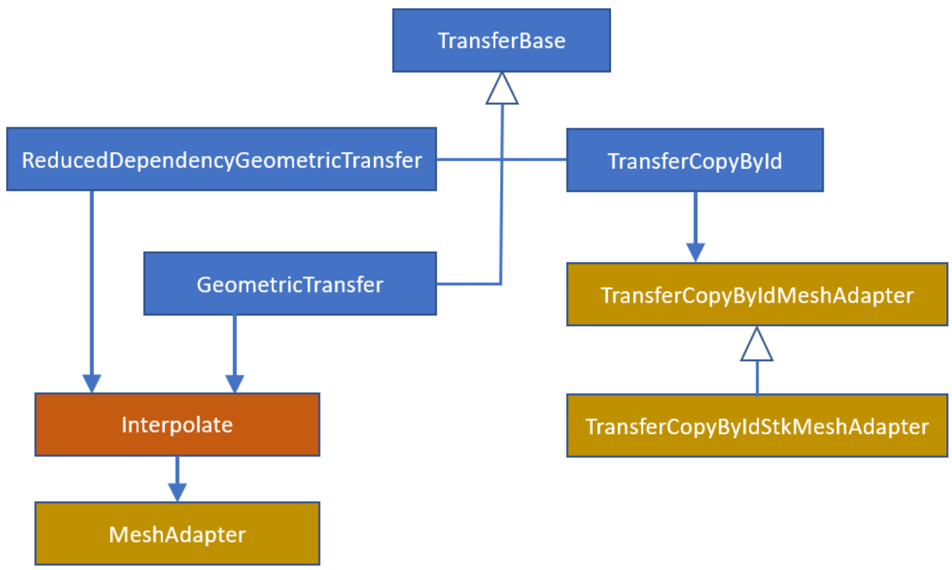



**Figure 8-1. STK Transfer class relationships**

The `TransferCopyById` class provides the ability to efficiently copy values between two identical meshes, independent of their parallel domain decomposition. Clients of this transfer capability must derive their own adapter class to interface with their mesh, which means that any mesh database may be used and there is no dependence on STK Mesh.

For cases where the source and destination meshes are not necessarily aligned or even when entirely different mesh databases are used, the `GeometricTransfer` and `ReducedDependencyGeometricTransfer` classes may be used. Both geometric transfer capabilities support Single-Program, Multiple-Data (SPMD) operation, while the `ReducedDependencyGeometricTransfer` adds the ability to function in a Multiple-Program, Multiple-Data (MPMD) context between two completely separate applications. Both of these transfer capabilities require clients to write an interpolation class and mesh adapter classes that are used as template parameters, giving the flexibility to perform any kind of interpolation between any two mesh databases. As with the copy transfer capability, there is no dependence on STK Mesh.

## 8.1. Copy Transfer

Copy transfers are used to copy field values between meshes that have the same geometry but potentially different parallel decomposition. Mesh entity IDs are used to identify matching source and destination pairs across all MPI ranks.

As shown in Figure 8-1, clients must implement an adapter class adhering to the interface provided by `TransferCopyByIdMeshAdapter`, that interfaces the transfer library with their specific mesh database so that it can get and set values correctly. For convenience, STK Transfer provides a `TransferCopyByIdStkMeshAdapter` implementation that can be used with instances of STK Mesh.

### *8.1.1. Copy Transfer Example with Geometric Search*

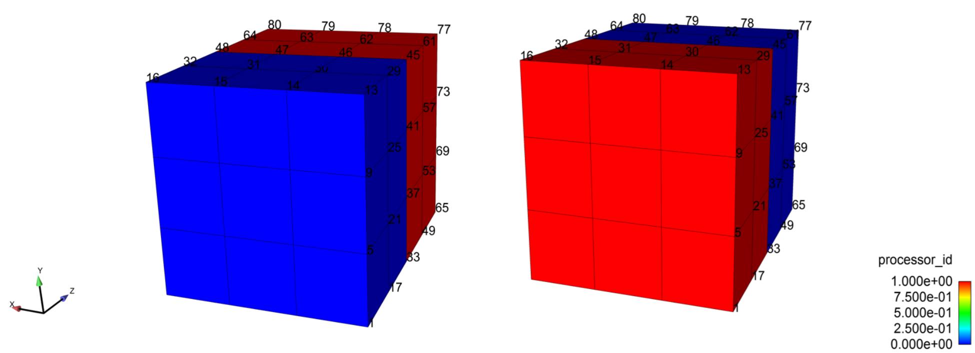



**Figure 8-2. Two identical meshes with different parallel decompositions**

Listing 8.1 shows an example of using `TransferCopyById` with a geometric search between two STK Meshes using `TransferCopyByIdStkMeshAdapter`. This example sets up two meshes, with each mesh having a different decomposition across the 2 MPI ranks as shown in Figure 8-2.

**Listing 8.1 Copy Transfer Example**
**code/stk/stk_doc_tests/stk_transfer/howToUseCopyTransfer.cpp**

```
TEST(StkTransferHowTo, useCopyTransfer)
{
  MPI_Comm communicator = MPI_COMM_WORLD;
  if (stk::parallel_machine_size(communicator) > 2) { GTEST_SKIP(); }

  const std::string meshSpec("generated:3x3x4");
  double initVals = std::numeric_limits<double>::max();
  const unsigned spatialDim = 3;

  stk::mesh::MeshBuilder builder(communicator);
  builder.set_spatial_dimension(spatialDim);
  std::shared_ptr<stk::mesh::BulkData> meshA = builder.create();
  stk::mesh::MetaData& metaA = meshA->mesh_meta_data();
  DoubleField & scalarFieldNodeA = metaA.declare_field<double>(stk::topology::NODE_RANK,
             "Node Scalar Field");
```

```
  stk::mesh::put_field_on_mesh(scalarFieldNodeA, metaA.universal_part(), &initVals);
  stk::io::fill_mesh(meshSpec, *meshA);

  std::shared_ptr<stk::mesh::BulkData> meshB = builder.create();
  stk::mesh::MetaData& metaB = meshB->mesh_meta_data();
  DoubleField & scalarFieldNodeB = metaB.declare_field<double>(stk::topology::NODE_RANK,
             "Node Scalar Field");
  stk::mesh::put_field_on_mesh(scalarFieldNodeB, metaB.universal_part(), &initVals);
  stk::io::fill_mesh(meshSpec, *meshB);

  change_mesh_decomposition(*meshB);

  set_field_vals_from_node_ids(*meshA, scalarFieldNodeA);

  // Set up CopyTransfer
  stk::mesh::EntityVector entitiesA;
  stk::mesh::get_entities(*meshA, stk::topology::NODE_RANK, metaA.locally_owned_part(),
             entitiesA);
  std::vector<stk::mesh::FieldBase*> fieldsA = {&scalarFieldNodeA};
  stk::transfer::TransferCopyByIdStkMeshAdapter transferMeshA(*meshA,entitiesA,fieldsA);

  stk::mesh::EntityVector entitiesB;
  stk::mesh::get_entities(*meshB, stk::topology::NODE_RANK, metaB.locally_owned_part(),
             entitiesB);
  std::vector<stk::mesh::FieldBase*> fieldsB = {&scalarFieldNodeB};
  stk::transfer::TransferCopyByIdStkMeshAdapter transferMeshB(*meshB,entitiesB,fieldsB);

  stk::transfer::SearchByIdGeometric copySearch;

  stk::transfer::TransferCopyById copyTransfer(copySearch,transferMeshA,transferMeshB);
  copyTransfer.initialize();
  copyTransfer.apply();

  // Verify nodal fields on meshB are correct
  stk::mesh::Selector owned = metaB.locally_owned_part();
  auto check_nodal_fields = [&scalarFieldNodeB](const stk::mesh::BulkData& mesh, const
             stk::mesh::Entity& node)
  {
    const double tolerance = 1.0e-8;
    double * scalar = stk::mesh::field_data(scalarFieldNodeB, node);
    EXPECT_NEAR( static_cast<double>(mesh.identifier(node)), *scalar, tolerance);
  };
  stk::mesh::for_each_entity_run(*meshB, stk::topology::NODE_RANK, owned, check_nodal_fields);
}
```

## 8.2. Geometric Transfer

The `GeometricTransfer` class is the next-most-general transfer capability available in STK after `TransferCopyById`. It can be used for interpolation transfers between unaligned source and destination meshes of any type, and is applicable only in an SPMD context where both the source and destination meshes exist in the same application. There is no requirement that the different meshes use the same set of MPI ranks or even that there is a good spatial correspondence in the parallel domain decompositions, although there will be a small performance enhancement due to reduced communication load if the source and destination mesh entities exist on the same MPI rank.

The overall idea of this transfer capability is that the receiving mesh provides a list of coordinates of discrete points at which it would like field data values. The sending mesh then interpolates or extrapolates the local field values, using whatever method is the most appropriate, to the requested coordinates from either local mesh entities or a copy of the source mesh entities from the originating MPI rank and copies the data into the receiving mesh.

### 8.2.1. *Example Geometric Transfer*

The generality of this transfer capability, where it can operate between any two mesh representations using any interpolation strategy, necessitates that users must write a significant amount of code to adapt the workflow to their specific needs. What follows is an example implementation of a highly-simplified transfer of two different data fields of different lengths between two instances of STK Mesh. The mesh database need not be the same on both sides of the transfer and the usage of STK Mesh is not required at all, although it is convenient for this demonstration.

Listing 8.2 shows a few supporting types that will be used throughout this example, and Listing 8.3 shows the main application. Two nodal fields are configured on each mesh – a scalar `temperature` field and a vector `velocity` field. The fields are given non-zero initial values on the sending mesh and zero initial values on the receiving mesh, so that we can easily detect a change once the transfer is complete. Both sides of the transfer should agree on the list of fields to be transferred to streamline processing. If the fields are not consistent, then the user must implement the ability to skip sending or receiving values that have no match on the other side. For simplicity, the fields are synchronized in this example.

**Listing 8.2 Supporting types to simplify geometric transfer example**
**code/stk/stk_doc_tests/stk_transfer/howToUseGeometricTransfer.cpp**

```
struct FieldConfigData {
  std::string name;
  stk::mesh::EntityRank rank;
  std::vector<double> initialValues;
};

using FieldConfig = std::vector<FieldConfigData>;
using BulkDataPtr = std::shared_ptr<stk::mesh::BulkData>;
```

**Listing 8.3 Main application for geometric transfer example**
**code/stk/stk_doc_tests/stk_transfer/howToUseGeometricTransfer.cpp**

```
template <typename INTERPOLATE>
using GeomTransfer = stk::transfer::GeometricTransfer<INTERPOLATE>;

using TransferType = GeomTransfer<Interpolate<StkSendAdapter, StkRecvAdapter>>;

std::shared_ptr<TransferType> setup_transfer(MPI_Comm globalComm,
                                             BulkDataPtr & sendBulk, BulkDataPtr & recvBulk,
                                             const FieldConfig & sendFieldConfig,
                                             const FieldConfig & recvFieldConfig)
{
  auto sendAdapter = std::make_shared<StkSendAdapter>(globalComm, sendBulk, "block_1",
                                                      sendFieldConfig);
  auto recvAdapter = std::make_shared<StkRecvAdapter>(globalComm, recvBulk, "block_1",
                                                      recvFieldConfig);

  auto transfer = std::make_shared<TransferType>(sendAdapter, recvAdapter,
```

```
                                                    "demoTransfer", globalComm);

  transfer->initialize();

  return transfer;
}

TEST(StkTransferHowTo, useGeometricTransfer)
{
  MPI_Comm commWorld = MPI_COMM_WORLD;

  FieldConfig sendFieldConfig {{"temperature", stk::topology::NODE_RANK, {300.0}},
                               {"velocity", stk::topology::NODE_RANK, {1.0, 2.0, 3.0}}};
  FieldConfig recvFieldConfig {{"temperature", stk::topology::NODE_RANK, {0.0}},
                               {"velocity", stk::topology::NODE_RANK, {0.0, 0.0, 0.0}}};

  BulkDataPtr sendBulk = read_mesh(commWorld, "generated:1x1x4", sendFieldConfig);
  BulkDataPtr recvBulk = read_mesh(commWorld, "generated:1x1x4", recvFieldConfig);

  auto transfer = setup_transfer(commWorld, sendBulk, recvBulk,
                                 sendFieldConfig, recvFieldConfig);

  transfer->apply();
  EXPECT_TRUE(all_field_values_equal(recvBulk, sendFieldConfig));
}
```

Both the sending and receiving meshes are read, and then both the coordinate field and the fields that will be transferred are initialized. This takes place in the `read_mesh()` function shown in Listing 8.4. For this example the meshes are identical and have the same parallel domain decomposition, although this is not a requirement.

**Listing 8.4 Supporting functions for geometric transfer example**
**code/stk/stk_doc_tests/stk_transfer/howToUseGeometricTransfer.cpp**

```
BulkDataPtr read_mesh(MPI_Comm comm,
                      const std::string & fileName,
                      const FieldConfig & fieldConfig)
{
  BulkDataPtr bulk = stk::mesh::MeshBuilder(comm).create();

  stk::io::StkMeshIoBroker ioBroker(comm);
  ioBroker.set_bulk_data(bulk);
  ioBroker.add_mesh_database(fileName, stk::io::READ_MESH);
  ioBroker.create_input_mesh();

  stk::mesh::MetaData& meta = bulk->mesh_meta_data();
  for (const FieldConfigData & fieldConf : fieldConfig) {
    auto & field = meta.declare_field<double>(fieldConf.rank, fieldConf.name);
    stk::mesh::put_field_on_mesh(field, meta.universal_part(), fieldConf.initialValues.size(),
                                 fieldConf.initialValues.data());
  }

  ioBroker.populate_bulk_data();

  return bulk;
}

bool all_field_values_equal(BulkDataPtr & bulk, const FieldConfig & fieldConfig)
{
  stk::mesh::MetaData & meta = bulk->mesh_meta_data();

  for (const FieldConfigData & fieldConf : fieldConfig) {
    const auto & field = *meta.get_field<double>(fieldConf.rank, fieldConf.name);
    stk::mesh::Selector fieldSelector(*meta.get_part("block_1"));
    const auto nodes = stk::mesh::get_entities(*bulk, fieldConf.rank, fieldSelector);
```

```
    for (stk::mesh::Entity node : nodes) {
      const double* fieldData = stk::mesh::field_data(field, node);
      for (unsigned i = 0; i < fieldConf.initialValues.size(); ++i) {
        if (std::abs(fieldData[i] - fieldConf.initialValues[i]) > 1.e-6) {
          return false;
        }
      }
    }
  }

  return true;
}
```

Next, the single transfer object for the whole application is constructed and configured in the `setup_transfer()` function, shown in Listing 8.3. This transfer object is an instance of `stk::transfer::GeometricTransfer<INTERPOLATE>` that is templated on a user-provided class that adheres to a specific interface, customized for managing the desired interpolation operations between the two meshes. The `INTERPOLATE` class itself may be templated on both a send-mesh adapter and a receive-mesh adapter class so that it can be compiled with knowledge of the appropriate types required to communicate with the two meshes. The `stk::transfer::GeometricTransfer` class has constructor arguments of a `std::shared_ptr` to instances of both the send-mesh adapter and the receive-mesh adapter, while the `INTERPOLATE` class is never constructed and must have its methods marked as **`static`** so that they may be called externally. Persistent information storage should take place on either the sending or receiving mesh adapters.

Once the transfer object is constructed, it is configured by making a call to its `initialize()` method. This is a shorthand for making sequential calls to the `coarse_search()`, `communication()`, and `local_search()` methods for the different stages of initial setup. The `coarse_search()` method internally uses STK Search (Chapter 7) to identify candidate mesh entities (elements, faces, etc.) on the sending side that correspond to the target coordinates on the receiving side. The `communication()` method then distributes lists of mesh entities on the sending side that must be copied to another processor to facilitate purely-local interpolation and copying of the result into the destination mesh. The `local_search()` method then identifies the best source mesh entity to interpolate data to each destination location and generates a unique one-to-one mapping between the meshes. User-provided supporting functions for each of these initialization calls will be discussed in the mesh adapter and `INTERPOLATE` class descriptions below.

This initial configuration work only needs to be done once if the meshes are static. If either mesh is modified or if entities in either mesh deform and change their coordinates, then this search and communication work will need to be re-done by calling `initialize()` again before the actual transfer operation occurs.

Once the transfer object has been constructed and configured, the application may trigger a data transfer at any time by calling `apply()`, as shown in Listing 8.3. This will do the actual data movement and interpolation on the sending side, followed by copying the results into the destination mesh.

This demonstration application has a final call to `all_field_values_equal()` (shown in Listing 8.4) on the receiving mesh to ensure that the transferred values get received and written correctly.

**Listing 8.5 Send-Mesh Adapter class for geometric transfer example**
**code/stk/stk_doc_tests/stk_transfer/howToUseGeometricTransfer.cpp**

```cpp
class StkSendAdapter
{
public:
  using EntityKey = stk::mesh::EntityKey;
  using EntityProc = stk::search::IdentProc<EntityKey, int>;
  using EntityProcVec = std::vector<EntityProc>;
  using BoundingBox = std::pair<stk::search::Box<double>, EntityProc>;

  using Coords = std::array<double, 3>;

  StkSendAdapter(MPI_Comm globalComm, BulkDataPtr & bulk,
                 const std::string & partName, const FieldConfig & fieldConfig)
    : m_globalComm(globalComm),
      m_bulk(bulk),
      m_meta(bulk->mesh_meta_data()),
      m_part(m_meta.get_part(partName)),
      m_ghosting(nullptr)
  {
    for (const FieldConfigData & fieldConf : fieldConfig) {
      m_fields.push_back(m_meta.get_field<double>(fieldConf.rank, fieldConf.name));
    }
  }

  MPI_Comm comm() const { return m_globalComm; }

  void bounding_boxes(std::vector<BoundingBox> & searchDomain) const
  {
    stk::mesh::Selector ownedSelector = m_meta.locally_owned_part() & *m_part;
    const auto elements = stk::mesh::get_entities(*m_bulk, stk::topology::ELEM_RANK,
                                                   ownedSelector);
    searchDomain.clear();
    const int procInSearchComm = stk::parallel_machine_rank(m_globalComm);
    for (stk::mesh::Entity element : elements) {
      EntityProc entityProc(m_bulk->entity_key(element), procInSearchComm);
      searchDomain.emplace_back(get_box(element), entityProc);
    }
  }

  void copy_entities(const EntityProcVec & entitiesToSend, const std::string & name)
  {
    m_ghostedEntities.clear();

    for (auto keyProc : entitiesToSend) {
      const stk::mesh::EntityKey key = keyProc.id();
      const unsigned proc = keyProc.proc();
      m_ghostedEntities.emplace_back(m_bulk->get_entity(key), proc);
    }

    unsigned hasEntitiesToGhost = not m_ghostedEntities.empty();

    stk::all_reduce(m_globalComm, stk::ReduceSum<1>(&hasEntitiesToGhost));

    if (hasEntitiesToGhost) {
      stk::util::sort_and_unique(m_ghostedEntities);

      m_bulk->modification_begin();
      if (m_ghosting == nullptr) {
        m_ghosting = &m_bulk->create_ghosting("transfer_ghosting");
      }
      m_bulk->change_ghosting(*m_ghosting, m_ghostedEntities);
      m_bulk->modification_end();
    }
```

```
  }

  void update_values()
  {
    std::vector<const stk::mesh::FieldBase*> commFields;
    for (stk::mesh::Field<double> * field : m_fields) {
      commFields.push_back(static_cast<stk::mesh::FieldBase*>(field));
    }
    stk::mesh::communicate_field_data(*m_bulk, commFields);
  }

  Coords parametric_coords(EntityKey entityKey, const double * spatialCoordinates,
                           double & distance) const
  {
    distance = 0.0;
    return Coords{0.0, 0.0, 0.0};
  }

  void interpolate_fields(const Coords & parametricCoords, EntityKey entityKey,
                          unsigned numFields, const std::vector<unsigned> & fieldSizes,
                          const std::vector<double *> & recvFieldPtrs) const
  {
    // This is where the actual application-specific shape function interpolation
    // operation would go.  For simplicity, this example uses zeroth-order
    // interpolation from only the first node's value.
    const stk::mesh::Entity targetElement = m_bulk->get_entity(entityKey);
    const stk::mesh::Entity firstNode = m_bulk->begin_nodes(targetElement)[0];
    for (unsigned n = 0; n < numFields; ++n) {
      const double * fieldData = stk::mesh::field_data(*m_fields[n], firstNode);
      for (unsigned idx = 0; idx < fieldSizes[n]; ++idx) {
        recvFieldPtrs[n][idx] = fieldData[idx];
      }
    }
  }

private:
  stk::search::Box<double> get_box(stk::mesh::Entity element) const
  {
    constexpr double minDouble = std::numeric_limits<double>::lowest();
    constexpr double maxDouble = std::numeric_limits<double>::max();
    double minXYZ[3] = {maxDouble, maxDouble, maxDouble};
    double maxXYZ[3] = {minDouble, minDouble, minDouble};
    const auto * coordField =
        static_cast<const stk::mesh::Field<double>*>(m_meta.coordinate_field());

    const stk::mesh::Entity * nodes = m_bulk->begin_nodes(element);
    const unsigned numNodes = m_bulk->num_nodes(element);
    for (unsigned i = 0; i < numNodes; ++i) {
      const double * coords = stk::mesh::field_data(*coordField, nodes[i]);
      minXYZ[0] = std::min(minXYZ[0], coords[0]);
      minXYZ[1] = std::min(minXYZ[1], coords[1]);
      minXYZ[2] = std::min(minXYZ[2], coords[2]);
      maxXYZ[0] = std::max(maxXYZ[0], coords[0]);
      maxXYZ[1] = std::max(maxXYZ[1], coords[1]);
      maxXYZ[2] = std::max(maxXYZ[2], coords[2]);
    }

    constexpr double tol = 1.e-5;
    return stk::search::Box<double>(minXYZ[0]-tol, minXYZ[1]-tol, minXYZ[2]-tol,
                                    maxXYZ[0]+tol, maxXYZ[1]+tol, maxXYZ[2]+tol);
  }

  MPI_Comm m_globalComm;
  BulkDataPtr m_bulk;
  stk::mesh::MetaData & m_meta;
  stk::mesh::Part* m_part;
  std::vector<stk::mesh::Field<double>*> m_fields;
  stk::mesh::Ghosting * m_ghosting;
```

```
  stk::mesh::EntityProcVec m_ghostedEntities;

};
```

The user must provide several supporting classes to the transfer library, including an adapter for the sending mesh, an adapter for the receiving mesh, and an overall interpolation class. We will look first at an example send-mesh adapter, shown in Listing 8.5. This is a class that provides a list of required types and class methods that will either be used directly by the `GeometricTransfer` class itself or your own `INTERPOLATE` class. This mesh adapter must provide definitions for the following types:

- `EntityKey`: This is an integral type that can be used as a unique global identifier for a mesh entity (e.g. element, face, node, etc.), and is used by your `INTERPOLATE` class to define other types for the core transfer library.
- `EntityProc`: This defines your customized `stk::search::IdentProc` type to pair together your unique global identifier for mesh entities and an MPI rank, and is used by your `INTERPOLATE` class to define another type for the core transfer library.
- `EntityProcVec`: This type defines a random-access container of your `EntityProc` types, and is expected to have an interface similar to `std::vector`. This is used by your `INTERPOLATE` class to define another type for the core transfer library.
- `BoundingBox`: This type is used directly by the core transfer library in conjunction with STK Search and defines a `std::pair` of a bounding box type from STK Search (usually something like `stk::search::Box<double>`) and your `EntityProc` type.

The `Coords` type defined in this example is not required by the transfer library, but is convenient for managing both spatial coordinates and parametric coordinates, and may be something as simple as a pointer to the start of a triplet of values in memory. A discrete type is used here for clarity.

There is no required signature for the constructor of this mesh adapter class, as client code will be constructing it directly and passing it into `GeometricTransfer`. The following class methods are required for a send-mesh adapter:

- `MPI_Comm comm() const`:
  This class method is used by the transfer library to retrieve the global MPI communicator used by your application.
- `void bounding_boxes(std::vector<BoundingBox> & searchDomain)`:
  This method will be called on the send-mesh adapter from `GeometricTransfer::coarse_search()` to get the full list of bounding boxes that contain all mesh entities that can be interpolated from. These may be boxes around things like elements (for a volumetric interpolation transfer) or faces (for a surface interpolation transfer) if something like shape function interpolation is going to be used, or it could even be boxes around individual nodes if something like a least-squares interpolation is going to be performed directly from a cloud of nodes. STK Search will be used to match these mesh entities up with target coordinate locations from the receiving side as candidates to provide the source data for interpolation.

- **void** `copy_entities(`**const** `EntityProcVec & entitiesToSend,` **const** `std::string & name):`
  This is an optional class method that will be called from the initial `communication()` function if it is provided. If the source and destination meshes are identical and have the same parallel domain decomposition, then the source data and the destination coordinates will exist on the same MPI rank and this function will not be necessary. Otherwise, this function will be called once it is determined which source mesh entities need to have their data copied to another processor so that interpolation and copying of the result into the destination mesh will be purely local operations. In STK Mesh, this is a ghosting operation (described in Section 4.6.2.7). Other mesh databases will need to provide an equivalent capability to mirror mesh data to another processor. The data attached to entities that are ghosted here will be updated in the `update_values()` method described below.

- **void** `update_values():`
  This method will be called at the start of the `GeometricTransfer::apply()` method to give the sending mesh a chance to synchronize field data to any ghosted mesh entities on different processors before it is used in an interpolation operation. In STK Mesh, updating field data on ghosted entities is done with a call to
  `stk::mesh::communicate_field_data().`

There are additional class methods implemented on the example send-mesh adapter that are not needed by the core transfer library, but are still likely to be needed by your `INTERPOLATE` class to service requests from the transfer library. These useful class methods are:

- `Coords parametric_coords(EntityKey entityKey,` **const double** `* spatialCoordinates,` **double** `& distance)` **const**:
  This function is needed when identifying which candidate mesh entity is the best to provide interpolated values to a destination location. For a given mesh entity and a spatial coordinate location, this function provides a parametric distance from the target location to the mesh entity centroid in addition to the actual parametric coordinates of this location within the mesh entity. The source fields are all uniform for this simple example, so it does not matter which mesh entity provides the result and all zeros are returned here.

- **void** `interpolate_fields(`**const** `Coords & parametricCoords, EntityKey entityKey,` **unsigned** `numFields,` **const** `std::vector<`**unsigned**`> & fieldSizes,` **const** `std::vector<`**double** `*> & recvFieldPtrs)` **const**:
  This function is responsible for performing the actual shape function interpolation. It is provided with a set of parametric coordinates and a source mesh entity within which the interpolation should be performed, as well as information about the fields that are to be interpolated. Pointers to the destination of the transfer will typically be provided so that the results of the interpolation can be written directly into the receiving mesh. This trivial example uses zeroth-order interpolation because the source fields are always uniform, meaning that we can simply select the first node of each element to provide the value. Your actual interpolation operation would go in this function.

**Listing 8.6 Receive-Mesh Adapter class for geometric transfer example code/stk/stk_doc_tests/stk_transfer/howToUseGeometricTransfer.cpp**

```
class StkRecvAdapter
{
public:
  using EntityKey = stk::mesh::EntityKey;
  using EntityProc = stk::search::IdentProc<EntityKey>;
  using EntityProcVec = std::vector<EntityProc>;
  using BoundingBox = std::pair<stk::search::Sphere<double>, EntityProc>;

  using Point = stk::search::Point<double>;
  using Coords = std::array<double, 3>;

  StkRecvAdapter(MPI_Comm globalComm, BulkDataPtr & bulk,
                 const std::string & partName, const FieldConfig & fieldConfig)
    : m_globalComm(globalComm),
      m_bulk(bulk),
      m_meta(m_bulk->mesh_meta_data()),
      m_part(m_meta.get_part(partName))
  {
    for (const FieldConfigData & fieldConf : fieldConfig) {
      m_fields.push_back(m_meta.get_field<double>(fieldConf.rank, fieldConf.name));
    }
  }

  MPI_Comm comm() const { return m_globalComm; }

  void bounding_boxes(std::vector<BoundingBox> & searchRange) const
  {
    stk::mesh::Selector ownedSelector = m_meta.locally_owned_part() & *m_part;
    const auto nodes = stk::mesh::get_entities(*m_bulk, stk::topology::NODE_RANK,
                                               ownedSelector);
    constexpr double radius = 1.e-6;
    searchRange.clear();
    const int procInSearchComm = stk::parallel_machine_rank(m_globalComm);
    for (const stk::mesh::Entity & node : nodes) {
      EntityProc entityProc(m_bulk->entity_key(node), procInSearchComm);
      searchRange.emplace_back(stk::search::Sphere<double>(get_location(node), radius),
                               entityProc);
    }
  }

  void update_values()
  {
    std::vector<const stk::mesh::FieldBase*> commFields;
    for (stk::mesh::Field<double> * field : m_fields) {
      commFields.push_back(static_cast<stk::mesh::FieldBase*>(field));
    }
    stk::mesh::communicate_field_data(*m_bulk, commFields);
  }

  const double * node_coords(EntityKey entityKey) const
  {
    const stk::mesh::Entity node = m_bulk->get_entity(entityKey);
    const auto & coordField =
        *static_cast<const stk::mesh::Field<double>*>(m_meta.coordinate_field());
    return stk::mesh::field_data(coordField, node);
  }

  void save_parametric_coords(EntityKey entityKey, const Coords & parametricCoords)
  {
    m_sendParametricCoords[entityKey] = parametricCoords;
  }

  unsigned num_fields() { return m_fields.size(); }

  const Coords & get_parametric_coords(EntityKey entityKey)
```

```
  {
    return m_sendParametricCoords.at(entityKey);
  }

  double * field_values(EntityKey entityKey, unsigned fieldIndex)
  {
    const stk::mesh::Entity node = m_bulk->get_entity(entityKey);
    return stk::mesh::field_data(*m_fields[fieldIndex], node);
  }

  unsigned field_size(EntityKey entityKey, unsigned fieldIndex)
  {
    const stk::mesh::Entity node = m_bulk->get_entity(entityKey);
    return stk::mesh::field_scalars_per_entity(*m_fields[fieldIndex], node);
  }

private:
  Point get_location(stk::mesh::Entity node) const
  {
    const auto & coordField =
        *static_cast<const stk::mesh::Field<double>*>(m_meta.coordinate_field());
    const double * coords = stk::mesh::field_data(coordField, node);

    return Point(coords[0], coords[1], coords[2]);
  }

  MPI_Comm m_globalComm;
  BulkDataPtr m_bulk;
  stk::mesh::MetaData & m_meta;
  stk::mesh::Part * m_part;
  std::vector<stk::mesh::Field<double>*> m_fields;
  std::map<EntityKey, Coords> m_sendParametricCoords;
};
```

---

The receive-mesh adapter, shown in Listing 8.6, is similar to the send-mesh adapter in that it encapsulates the receiving mesh and provides a list of required types and class methods for use by either your `INTERPOLATE` class or the core transfer library itself. This mesh adapter must provide definitions for the following types:

- `EntityKey`: This is an integral type that can be used as a unique global identifier for a mesh entity (e.g. element, face, node, etc.), and is used by your `INTERPOLATE` class to define other types for the core transfer library.
- `EntityProc`: This defines your customized `stk::search::IdentProc` type to pair together your unique global identifier for mesh entities and an MPI rank, and is used by your `INTERPOLATE` class to define other types for the core transfer library.
- `EntityProcVec`: This type defines a random-access container of your `EntityProc` types, and is expected to have an interface similar to `std::vector`. This is used by your `INTERPOLATE` class to define other types for the core transfer library.
- `BoundingBox`: This type is used directly by the core transfer library in conjunction with STK Search and defines a `std::pair` of a bounding box type from STK Search (usually something like `stk::search::Sphere<double>`) to define the location of your target interpolation coordinates, and your `EntityProc` type.

The `Point` type defined in this example is not required by the transfer library, but is convenient when building a `stk::search::Sphere`. The `Coords` type defined in this example is also not required by the transfer library, but is convenient for managing both spatial coordinates and parametric coordinates, and may be something as simple as a pointer to the start of a triplet of values in memory. A discrete type is used here for clarity.

As with the send-mesh adapter, there is no required signature for the receive-mesh adapter constructor. The following class methods are required for a receive-mesh adapter:

- `MPI_Comm comm() const`:
  This class method is used by the transfer library to retrieve the global MPI communicator used by your application.
- `void bounding_boxes(std::vector<BoundingBox> & searchRange)`:
  This method will be called from the `GeometricTransfer::coarse_search()` method to get the full list of bounding boxes that contain each of the discrete coordinates that the field values will be interpolated to. STK Search will be used to match these coordinates up with candidate mesh entities on the sending mesh that will be used to interpolate the results to be transferred.
- `void update_values()`:
  This method will be called at the end of the `GeometricTransfer::apply()` method to give the receive-mesh adapter a chance to do any cleanup work after receiving the interpolated data, such as possibly updating field values on any shared entities along parallel boundaries. This method may be empty if there is no work to do.

As with the send-mesh adapter, there are several class methods implemented on the receive-mesh adapter here that are not required by the core transfer library, but are still likely to be needed by your `INTERPOLATE` class to service requests from the transfer library. These useful class methods are:

- `const double * node_coords(EntityKey entityKey) const`:
  This class method provides the discrete coordinates of a node, which is useful when filtering entities on the sending mesh to determine which can provide the highest-quality interpolated value.
- `void save_parametric_coords(EntityKey entityKey, const Coords & parametricCoords)`:
  When your `INTERPOLATE` class is filtering the mesh entities on the sending mesh to find the one that can provide the highest-quality interpolated result, it can call this function to store the parametric coordinates within that mesh entity that will be used to interpolate a value for the provided receive-mesh entity. It might make more logical sense to store this data on the send-mesh adapter where it will be used, although it is not a unique one-to-one mapping like it is on the receiving mesh.
- `unsigned num_fields()`:
  This method provides the number of fields that need to be interpolated, for use in sizing various data arrays at interpolation time.
- `double * field_values(EntityKey entityKey, unsigned fieldIndex)`:
  This method acquires a pointer to the destination of the interpolation for a particular mesh

entity. The interpolated results will be written directly to this memory location.

- `unsigned field_size(EntityKey entityKey, unsigned fieldIndex)`:
  This method provides the number of scalars that will be written into the destination mesh for a particular field

**Listing 8.7 Interpolation class for geometric transfer example**
**code/stk/stk_doc_tests/stk_transfer/howToUseGeometricTransfer.cpp**

```
template<typename SendAdapter, typename RecvAdapter>
class Interpolate
{
public:
  using MeshA = SendAdapter;
  using MeshB = RecvAdapter;
  using EntityKeyA = typename MeshA::EntityKey;
  using EntityKeyB = typename MeshB::EntityKey;
  using EntityProcA = typename MeshA::EntityProc;
  using EntityProcB = typename MeshB::EntityProc;
  using EntityKeyMap = std::multimap<EntityKeyB, EntityKeyA>;
  using EntityProcRelation = std::pair<EntityProcB, EntityProcA>;
  using EntityProcRelationVec = std::vector<EntityProcRelation>;

  using Coords = typename MeshA::Coords;

  static void filter_to_nearest(EntityKeyMap & localRangeToDomain,
                                MeshA & sendMesh, MeshB & recvMesh)
  {
    using iterator = typename EntityKeyMap::iterator;
    using const_iterator = typename EntityKeyMap::const_iterator;

    for (const_iterator key = localRangeToDomain.begin(); key != localRangeToDomain.end();)
    {
      const EntityKeyB recvEntityKey = key->first;
      double closestDistance = std::numeric_limits<double>::max();

      const double * recvCoords = recvMesh.node_coords(recvEntityKey);

      std::pair<iterator, iterator> sendEntities =
            localRangeToDomain.equal_range(recvEntityKey);
      iterator nearest = sendEntities.second;

      for (iterator ii = sendEntities.first; ii != sendEntities.second; ++ii) {
        const EntityKeyA sendEntity = ii->second;

        double distance = 0;
        const Coords parametricCoords = sendMesh.parametric_coords(sendEntity, recvCoords,
                                                                   distance);

        if (distance < closestDistance) {
          closestDistance = distance;
          recvMesh.save_parametric_coords(recvEntityKey, parametricCoords);
          nearest = ii;
        }
      }

      key = sendEntities.second;
      if (nearest != sendEntities.first) {
        localRangeToDomain.erase(sendEntities.first, nearest);
      }
      if (nearest != sendEntities.second) {
        localRangeToDomain.erase(++nearest, sendEntities.second);
      }
    }
  }
```

```
  static void apply(MeshB & recvMesh, MeshA & sendMesh, EntityKeyMap & localRangeToDomain)
  {
    const unsigned numFields = recvMesh.num_fields();

    std::vector<double *> fieldPtrs(numFields);
    std::vector<unsigned> fieldSizes(numFields);

    typename EntityKeyMap::const_iterator ii;
    for (ii = localRangeToDomain.begin(); ii != localRangeToDomain.end(); ++ii) {
      const EntityKeyB recvNode = ii->first;
      const EntityKeyA sendElem = ii->second;

      const Coords & sendParametricCoords = recvMesh.get_parametric_coords(recvNode);

      for (unsigned n = 0; n < numFields; ++n) {
        fieldPtrs[n] = recvMesh.field_values(recvNode, n);
        fieldSizes[n] = recvMesh.field_size(recvNode, n);
      }

      sendMesh.interpolate_fields(sendParametricCoords, sendElem, numFields,
                                  fieldSizes, fieldPtrs);
    }
  }

};
```

The `INTERPOLATE` class template parameter for the transfer object will be discussed next. This class manages entity selection during the initial setup as well as the actual interpolation and data movement at run-time. A simple example is the `Interpolate` class shown in Listing 8.7. As with the mesh adapter classes, a number of types must be defined to satisfy the requirements of the `GeometricTransfer` library. These types are as follows:

- `MeshA`: This is the type of the send-mesh adapter, and is used by the core transfer library to directly interrogate types provided by this mesh adapter in order to communicate with it.
- `MeshB`: This is the type of the receive-mesh adapter, and is used by the core transfer library to directly interrogate types provided by this mesh adapter in order to communicate with it. The sending and receiving meshes need not be of the same type.
- `EntityKeyA`: This is the `EntityKey` type used by the send-mesh adapter, which is an integral type intended to be used as a unique global identifier for mesh entities.
- `EntityKeyB`: This is the `EntityKey` type used by the receive-mesh adapter, which is an integral type intended to be used as a unique global identifier for mesh entities. The two `EntityKey` types need not be the same.
- `EntityProcA`: This is the customized `stk::search::IdentProc` type defined by your send-mesh adapter, and is used to bundle together your unique global identifier for a mesh entity and an MPI processor rank.
- `EntityProcB`: This is the customized `stk::search::IdentProc` type defined by your receive-mesh adapter, and is used to bundle together your unique global identifier for a mesh entity and an MPI processor rank.
- `EntityKeyMap`: This is an associative container type that is capable of matching multiple `EntityKeyA` entries with each `EntityKeyB`, and is expected to have an API similar to a

`std::multimap`. This is the type of the container that holds the results of the initial STK Search to match all candidate mesh entities in the sending mesh with each location on the receiving mesh.

- `EntityProcRelation`: This is a pair of `EntityProcB` and `EntityProcA` types in a container that is equivalent to `std::pair`, and is used to store relationships between a unique pair of mesh entities on the sending and receiving meshes.
- `EntityProcRelationVec`: This is a random-access container of `EntityProcRelation` types, equivalent to a `std::vector`.

The `INTERPOLATE` class will not be instantiated by the `GeometricTransfer` library, so the required class methods must be marked **`static`** so that they may be called. Below are listed the methods that this class must provide:

- **`static void`** `filter_to_nearest(EntityKeyMap & localRangeToDomain, MeshA & sendMesh, MeshB & recvMesh):`
  This function is called from `GeometricTransfer::local_search()` to narrow down the list of all candidate mesh entities on the sending side to isolate the single best entity to provide an interpolated value for each coordinate location on the receiving mesh. A reasonable criterion for selecting the best entity would be to minimize the parametric distance of the target coordinate from the entity centroid, although any measure of interpolation quality may be used. Once the best entity is selected for each target coordinate, you are expected to remove all other entities from the provided container so that only a unique one-to-one mapping is retained. This will be the final list of mesh entities used during interpolation. It might also be desirable to store the parametric coordinates of each target location while it is available, to streamline the actual interpolation operations at run-time.
- **`static void`** `apply(MeshB & recvMesh, MeshA & sendMesh, EntityKeyMap & localRangeToDomain):`
  This is the main function to perform the interpolation operation at run-time, and is called from `GeometricTransfer::apply()` after any remote field data is copied to the local processor in `MeshB::update_values()`. As a result of this data movement, this function can perform purely local operations. You will be given pairs of mesh entities on the sending and receiving meshes, and you are expected to interpolate all of the desired field values from the sending mesh and copy the results into the receiving mesh.

### 8.3. Reduced-Dependency Geometric Transfer

The `ReducedDependencyGeometricTransfer` class is the most general transfer capability available in STK. It can be used for interpolation transfers between unaligned source and destination meshes of any type, and it can be used in either a Single-Program, Multiple-Data (SPMD) or a Multiple-Program, Multiple-Data (MPMD) context, giving the flexibility of transferring data between two meshes in a single application or two meshes in completely separate applications that are launched in a single MPI context, respectively.

The overall idea is that the receiving mesh provides a list of coordinates of discrete points at which it would like field data values. The sending mesh then interpolates or extrapolates the local field values to the requested coordinates using whatever method is most appropriate, and communicates the data to the receiving mesh.

Usage of this transfer capability will be illustrated for both an SPMD and an MPMD context in the following two sub-sections.

### 8.3.1. Example SPMD Reduced-Dependency Geometric Transfer

The extreme generality of this transfer capability, where it can operate between any two applications or within a single application, and between any two mesh representations, necessitates that users must write a significant amount of code to adapt the workflow to their specific needs. The work needed to interface with this transfer capability is somewhat simpler than what is needed for the `GeometricTransfer` capability, mostly due to the transfer library itself taking over some of the prior tasks and performing them more generically. The `INTERPOLATE` class now processes generic data from each mesh adapter instead of directly manipulating the mesh adapters, which helps with dependency isolation between separate applications.

What follows is an example of a highly-simplified transfer of two different data fields of different lengths between two instances of STK Mesh within the same application. Again, the mesh database need not be the same on both sides of the transfer and the usage of STK Mesh is not required, although it is convenient for this demonstration.

Listing 8.8 shows a few supporting types that will be used throughout this example, and Listing 8.9 shows the main application. Two nodal fields are configured on both meshes – a scalar `temperature` field and a vector `velocity` field. The fields are given non-zero initial values on the sending side and zero initial values on the receiving side, so that we can easily detect a change once the transfer is complete. Both sides of the transfer must agree on the list of fields so that they can properly encode/decode the serialized MPI data stream that packs all of the interpolated values together.

**Listing 8.8 Supporting types to simplify reduced-dependency geometric transfer examples**
**code/stk/stk_doc_tests/stk_transfer/howToUseReducedDependencyGeometricTransfer.cpp**

```
struct FieldConfigData {
  std::string name;
  stk::mesh::EntityRank rank;
  std::vector<double> initialValues;
};

using FieldConfig = std::vector<FieldConfigData>;
using BulkDataPtr = std::shared_ptr<stk::mesh::BulkData>;
```

**Listing 8.9 Main application for SPMD reduced-dependency geometric transfer example**
**code/stk/stk_doc_tests/stk_transfer/howToUseReducedDependencyGeometricTransfer.cpp**

```
template <typename INTERPOLATE>
using RDGeomTransfer = stk::transfer::ReducedDependencyGeometricTransfer<INTERPOLATE>;

using TransferType = RDGeomTransfer<Interpolate<StkSendAdapter, StkRecvAdapter>>;

```

```
std::shared_ptr<TransferType> setup_transfer(MPI_Comm globalComm,
                                             BulkDataPtr & sendBulk,
                                             BulkDataPtr & recvBulk,
                                             const FieldConfig & sendFieldConfig,
                                             const FieldConfig & recvFieldConfig)
{
  auto sendAdapter = std::make_shared<StkSendAdapter>(globalComm, sendBulk, "block_1",
            sendFieldConfig);
  auto recvAdapter = std::make_shared<StkRecvAdapter>(globalComm, recvBulk, "block_1",
            recvFieldConfig);

  auto transfer = std::make_shared<TransferType>(sendAdapter, recvAdapter, "demoTransfer",
            globalComm);

  transfer->initialize();

  return transfer;
}

TEST(StkTransferHowTo, useReducedDependencyGeometricTransferSPMD)
{
  MPI_Comm commWorld = MPI_COMM_WORLD;

  FieldConfig sendFieldConfig {{"temperature", stk::topology::NODE_RANK, {300.0}},
                               {"velocity", stk::topology::NODE_RANK, {1.0, 2.0, 3.0}}};
  FieldConfig recvFieldConfig {{"temperature", stk::topology::NODE_RANK, {0.0}},
                               {"velocity", stk::topology::NODE_RANK, {0.0, 0.0, 0.0}}};

  BulkDataPtr sendBulk = read_mesh(commWorld, "generated:1x1x4", sendFieldConfig);
  BulkDataPtr recvBulk = read_mesh(commWorld, "generated:1x1x4", recvFieldConfig);

  auto transfer = setup_transfer(commWorld, sendBulk, recvBulk, sendFieldConfig,
            recvFieldConfig);

  transfer->apply();
  EXPECT_TRUE(all_field_values_equal(recvBulk, sendFieldConfig));
}
```

Both the sending and receiving meshes are read and then the coordinate field and the fields that will be transferred are initialized. This takes place in the `read_mesh()` function shown in Listing 8.10. For this example the meshes are identical, although they are not required to be. The only requirement is that the spatial coordinates have some commonality between the two meshes so that matching locations can be identified between the sending and receiving sides.

**Listing 8.10 Supporting functions for reduced-dependency geometric transfer examples**
**code/stk/stk_doc_tests/stk_transfer/howToUseReducedDependencyGeometricTransfer.cpp**

```
BulkDataPtr read_mesh(MPI_Comm comm,
                      const std::string & fileName,
                      const FieldConfig & fieldConfig)
{
  BulkDataPtr bulk = stk::mesh::MeshBuilder(comm).create();

  stk::io::StkMeshIoBroker ioBroker(comm);
  ioBroker.set_bulk_data(bulk);
  ioBroker.add_mesh_database(fileName, stk::io::READ_MESH);
  ioBroker.create_input_mesh();

  stk::mesh::MetaData& meta = bulk->mesh_meta_data();
  for (const FieldConfigData & fieldConf : fieldConfig) {
    auto & field = meta.declare_field<double>(fieldConf.rank, fieldConf.name);
    stk::mesh::put_field_on_mesh(field, meta.universal_part(), fieldConf.initialValues.size(),
                                 fieldConf.initialValues.data());
  }
```

```
  ioBroker.populate_bulk_data();

  return bulk;
}

bool all_field_values_equal(BulkDataPtr & bulk, const FieldConfig & fieldConfig)
{
  stk::mesh::MetaData & meta = bulk->mesh_meta_data();

  for (const FieldConfigData & fieldConf : fieldConfig) {
    const auto & field = *meta.get_field<double>(fieldConf.rank, fieldConf.name);
    stk::mesh::Selector fieldSelector(*meta.get_part("block_1"));
    const auto nodes = stk::mesh::get_entities(*bulk, fieldConf.rank, fieldSelector);
    for (stk::mesh::Entity node : nodes) {
      const double* fieldData = stk::mesh::field_data(field, node);
      for (unsigned i = 0; i < fieldConf.initialValues.size(); ++i) {
        if (std::abs(fieldData[i] - fieldConf.initialValues[i]) > 1.e-6) {
          return false;
        }
      }
    }
  }

  return true;
}
```

Next, the single transfer object for the whole application is constructed and configured in the `setup_transfer()` function, shown in Listing 8.9. This transfer object is an instance of `stk::transfer::ReducedDependencyGeometricTransfer<INTERPOLATE>` that is templated on a user-provided class that adheres to a specific interface, customized for managing the desired interpolation operations between the two meshes. The `INTERPOLATE` class itself may be templated on both a send-mesh adapter and a receive-mesh adapter class so that it can be compiled with knowledge of the appropriate types required to communicate with the two meshes. The `stk::transfer::ReducedDependencyGeometricTransfer` class has constructor arguments of a `std::shared_ptr` to instances of both the send-mesh adapter and the receive-mesh adapter, while the `INTERPOLATE` class is default-constructed internally.

Once the transfer object is constructed, it is configured by making a call to its `initialize()` method. This is a shorthand for making sequential calls to `coarse_search()` and `communication()` for the different stages of initial setup. A call is also made to `local_search()`, but this class method does nothing for this transfer capability. The `coarse_search()` method internally uses STK Search (Chapter 7) to identify candidate mesh entities (elements, faces, etc.) on the sending side that correspond to the target coordinates on the receiving side. The `communication()` method then shares the lists of mesh entities among the processors and identifies an optimal set of unique one-to-one mappings between the two meshes. User-provided supporting functions for each of these initialization calls will be discussed in the mesh adapter and `INTERPOLATE` class descriptions below.

This initial configuration work only needs to be done once if the meshes are static. If either mesh is modified or if entities in either mesh deform and change their coordinates, then this search and communication work will need to be re-done before the actual transfer operation occurs, to maintain accuracy and correctness.

Once the transfer object has been constructed and configured, the applications may trigger a data transfer at any time by calling `apply()` on the transfer object, as shown in Listing 8.9. This will do the actual interpolation on the sending side, package it up, and send it over to the receiving side where it can be inserted into the destination mesh.

This demonstration application has a final call to `all_field_values_equal()` on the receiving mesh to ensure that the transferred values get received and written correctly.

**Listing 8.11 Send-Mesh Adapter class for reduced-dependency geometric transfer example**
**code/stk/stk_doc_tests/stk_transfer/howToUseReducedDependencyGeometricTransfer.cpp**

```
class StkSendAdapter
{
public:
  using EntityKey = stk::mesh::EntityKey;
  using EntityProc = stk::search::IdentProc<EntityKey, int>;
  using EntityProcVec = std::vector<EntityProc>;

  using BoundingBox = std::pair<stk::search::Box<double>, EntityProc>;

  StkSendAdapter(MPI_Comm globalComm, BulkDataPtr & bulk,
                 const std::string & partName, const FieldConfig & fieldConfig)
    : m_globalComm(globalComm),
      m_bulk(bulk),
      m_meta(bulk->mesh_meta_data()),
      m_part(m_meta.get_part(partName))
  {
    unsigned totalFieldSize = 0;
    for (const FieldConfigData & fieldConf : fieldConfig) {
      m_fields.push_back(m_meta.get_field<double>(fieldConf.rank, fieldConf.name));
      totalFieldSize += fieldConf.initialValues.size();
    }
    m_totalFieldSize = totalFieldSize;
  }

  void bounding_boxes(std::vector<BoundingBox> & searchDomain) const
  {
    stk::mesh::Selector ownedSelector = m_meta.locally_owned_part() & *m_part;
    const auto elements = stk::mesh::get_entities(*m_bulk, stk::topology::ELEM_RANK,
                                                  ownedSelector);
    searchDomain.clear();
    const int procInSearchComm = stk::parallel_machine_rank(m_globalComm);
    for (stk::mesh::Entity element : elements) {
      EntityProc entityProc(m_bulk->entity_key(element), procInSearchComm);
      searchDomain.emplace_back(get_box(element), entityProc);
    }
  }

  void update_values()
  {
    std::vector<const stk::mesh::FieldBase*> commFields;
    for (stk::mesh::Field<double> * field : m_fields) {
      commFields.push_back(static_cast<stk::mesh::FieldBase*>(field));
    }
    stk::mesh::communicate_field_data(*m_bulk, commFields);
  }

  void interpolate_fields(const std::array<double, 3> & parametricCoords,
                          EntityKey entityKey, double * interpValues) const
  {
    // This is where the actual application-specific shape function interpolation
    // operation would go.  For simplicity, this example uses zeroth-order
    // interpolation from only the first node's value.
    const stk::mesh::Entity targetElement = m_bulk->get_entity(entityKey);
    const stk::mesh::Entity firstNode = m_bulk->begin_nodes(targetElement)[0];
    unsigned offset = 0;
    for (const stk::mesh::Field<double> * field : m_fields) {
```

```
      const double * fieldData = stk::mesh::field_data(*field, firstNode);
      for (unsigned idx = 0; idx < field->max_size(); ++idx) {
        interpValues[offset++] = fieldData[idx];
      }
    }
  }

  unsigned total_field_size() const { return m_totalFieldSize; }

private:
  stk::search::Box<double> get_box(stk::mesh::Entity element) const
  {
    constexpr double minDouble = std::numeric_limits<double>::lowest();
    constexpr double maxDouble = std::numeric_limits<double>::max();
    double minXYZ[3] = {maxDouble, maxDouble, maxDouble};
    double maxXYZ[3] = {minDouble, minDouble, minDouble};
    const auto * coordField =
        static_cast<const stk::mesh::Field<double>*>(m_meta.coordinate_field());

    const stk::mesh::Entity * nodes = m_bulk->begin_nodes(element);
    const unsigned numNodes = m_bulk->num_nodes(element);
    for (unsigned i = 0; i < numNodes; ++i) {
      const double * coords = stk::mesh::field_data(*coordField, nodes[i]);
      minXYZ[0] = std::min(minXYZ[0], coords[0]);
      minXYZ[1] = std::min(minXYZ[1], coords[1]);
      minXYZ[2] = std::min(minXYZ[2], coords[2]);
      maxXYZ[0] = std::max(maxXYZ[0], coords[0]);
      maxXYZ[1] = std::max(maxXYZ[1], coords[1]);
      maxXYZ[2] = std::max(maxXYZ[2], coords[2]);
    }

    constexpr double tol = 1.e-5;
    return stk::search::Box<double>(minXYZ[0]-tol, minXYZ[1]-tol, minXYZ[2]-tol,
                                    maxXYZ[0]+tol, maxXYZ[1]+tol, maxXYZ[2]+tol);
  }

  MPI_Comm m_globalComm;
  BulkDataPtr m_bulk;
  stk::mesh::MetaData & m_meta;
  stk::mesh::Part* m_part;
  std::vector<stk::mesh::Field<double>*> m_fields;
  unsigned m_totalFieldSize;
};
```

To construct the transfer object, users need to implement a mesh adapter class for both the sending mesh and the receiving mesh, as well as an interpolation class that manages the data movement and communication between the two meshes. We will look first at an example send-mesh adapter, shown in Listing 8.11. This is a class that provides a list of required types and class methods that will either be used directly by the `ReducedDependencyGeometricTransfer` class itself or your own `INTERPOLATE` class. This mesh adapter must provide definitions for the following types:

- `EntityKey`: This is an integral type that can be used as a unique global identifier for a mesh entity (e.g. element, face, node, etc.), and is used by your interpolation class to define other types for the core transfer library.
- `EntityProc`: This defines your customized `stk::search::IdentProc` type to pair together your unique global identifier for mesh entities and an MPI processor rank, and is used by your interpolation class to define another type for the core transfer library.
- `EntityProcVec`: This type defines a random-access container of your `EntityProc`

types, and is expected to have an interface similar to `std::vector`. This is used by your interpolation class to define another type for the core transfer library.

- `BoundingBox`: This type is used directly by the core transfer library in conjunction with STK Search and defines a `std::pair` of a bounding box type from STK Search (usually something like `stk::search::Box<double>`) and your `EntityProc` type.

There is no required signature for the constructor of this class, as client code will be constructing it directly and passing it into `ReducedDependencyGeometricTransfer`. The following class methods are required for a send-mesh adapter:

- `void bounding_boxes(std::vector<BoundingBox> & searchDomain)`: This class method will be called from `ReducedDependencyGeometricTransfer::coarse_search()` to get the full list of bounding boxes that contain all mesh entities that can be interpolated from. These may be boxes around things like elements (for a volumetric interpolation transfer) or faces (for a surface interpolation transfer) if something like shape function interpolation is going to be used, or it could even be boxes around individual nodes if something like a least-squares interpolation is going to be performed directly from the nodes. STK Search will be used to match these mesh entities up with coordinate locations from the receiving mesh to determine the best one-to-one mapping of a single mesh entity on the sending side with each coordinate on the receiving side.
- `void update_values()`: This method will be called on the send-mesh adapter at the start of `ReducedDependencyGeometricTransfer::apply()` to give the sending mesh a chance to do any preparation work before interpolating the data, such as possibly updating field values on any shared entities along parallel boundaries. This method may be empty if there is no work to do.

There are additional class methods implemented on the example send-mesh adapter that are not needed by the core transfer library, but are still likely to be needed by your `INTERPOLATE` class to service requests from the transfer library. These useful class methods are:

- `void interpolate_fields(const std::array<double, 3> & parametricCoords, EntityKey entityKey, double * interpValues) const`: This class will typically be provided with a set of parametric coordinates within the source mesh entity to be interpolated from, as well as an `EntityKey` to identify the specific mesh entity. This method needs to perform the actual field data interpolation to the specified location and place the results for all fields into a compact row of data that will be communicated to the receiving processor.
- `unsigned total_field_size() const`: This method provides the total number of `double` values summed across all fields that are being interpolated, to help with striding through the data that will be communicated.

**Listing 8.12 Receive-Mesh Adapter class for reduced-dependency geometric transfer example code/stk/stk_doc_tests/stk_transfer/howToUseReducedDependencyGeometricTransfer.cpp**

```
class StkRecvAdapter
{
```

```
public:
  using EntityKey = stk::mesh::EntityKey;
  using EntityProc = stk::search::IdentProc<EntityKey>;
  using EntityProcVec = std::vector<EntityProc>;

  using BoundingBox = std::pair<stk::search::Sphere<double>, EntityProc>;
  using Point = stk::search::Point<double>;
  using ToPointsContainer = std::vector<Point>;
  using ToPointsDistanceContainer = std::vector<double>;

  StkRecvAdapter(MPI_Comm globalComm, BulkDataPtr & bulk,
                 const std::string & partName, const FieldConfig & fieldConfig)
    : m_globalComm(globalComm),
      m_bulk(bulk),
      m_meta(m_bulk->mesh_meta_data()),
      m_part(m_meta.get_part(partName))
  {
    unsigned totalFieldSize = 0;
    for (const FieldConfigData & fieldConf : fieldConfig) {
      m_fields.push_back(m_meta.get_field<double>(fieldConf.rank, fieldConf.name));
      totalFieldSize += fieldConf.initialValues.size();
    }
    m_totalFieldSize = totalFieldSize;
  }

  void bounding_boxes(std::vector<BoundingBox> & searchRange) const
  {
    stk::mesh::Selector ownedSelector = m_meta.locally_owned_part() & *m_part;
    const auto nodes = stk::mesh::get_entities(*m_bulk, stk::topology::NODE_RANK,
                                               ownedSelector);
    constexpr double radius = 1.e-6;
    searchRange.clear();
    const int procInSearchComm = stk::parallel_machine_rank(m_globalComm);
    for (const stk::mesh::Entity & node : nodes) {
      EntityProc entityProc(m_bulk->entity_key(node), procInSearchComm);
      searchRange.emplace_back(stk::search::Sphere<double>(get_location(node), radius),
                               entityProc);
    }
  }

  void get_to_points_coordinates(const EntityProcVec & toEntityKeys,
                                 ToPointsContainer & toPoints)
  {
    toPoints.clear();
    for (EntityProc entityProc : toEntityKeys) {
      toPoints.push_back(get_location(m_bulk->get_entity(entityProc.id())));
    }
  }

  void update_values()
  {
    std::vector<const stk::mesh::FieldBase*> commFields;
    for (stk::mesh::Field<double> * field : m_fields) {
      commFields.push_back(static_cast<stk::mesh::FieldBase*>(field));
    }
    stk::mesh::communicate_field_data(*m_bulk, commFields);
  }

  void set_field_values(const EntityKey & entityKey, const double * recvInterpValues)
  {
    stk::mesh::Entity node = m_bulk->get_entity(entityKey);
    unsigned offset = 0;
    for (const stk::mesh::Field<double> * field : m_fields) {
      double * fieldData = stk::mesh::field_data(*field, node);
      for (unsigned idx = 0; idx < field->max_size(); ++idx) {
        fieldData[idx] = recvInterpValues[offset++];
      }
    }
```

```
  }

  unsigned total_field_size() const { return m_totalFieldSize; }

private:
  Point get_location(stk::mesh::Entity node) const
  {
    const auto & coordField =
        *static_cast<const stk::mesh::Field<double>*>(m_meta.coordinate_field());
    const double * coords = stk::mesh::field_data(coordField, node);

    return Point(coords[0], coords[1], coords[2]);
  }

  MPI_Comm m_globalComm;
  BulkDataPtr m_bulk;
  stk::mesh::MetaData & m_meta;
  stk::mesh::Part * m_part;
  std::vector<stk::mesh::Field<double>*> m_fields;
  unsigned m_totalFieldSize;
};
```

The receive-mesh adapter, shown in Listing 8.12, is similar to the send-mesh adapter in that it encapsulates the receiving mesh and provides a list of required types and class methods for use by either your `INTERPOLATE` class or the core transfer library itself. This mesh adapter must provide definitions for the following types:

- `EntityKey`: This is an integral type that can be used as a unique global identifier for a mesh entity (e.g. element, face, node, etc.), and is used by your interpolation class to define other types for the core transfer library.
- `EntityProc`: This defines your customized `stk::search::IdentProc` type to pair together your unique global identifier for mesh entities and an MPI processor rank, and is used by your interpolation class to define other types for the core transfer library.
- `EntityProcVec`: This type defines a random-access container of your `EntityProc` types, and is expected to have an interface similar to `std::vector`. This is used by your interpolation class to define other types for the core transfer library.
- `BoundingBox`: This type is used directly by the core transfer library in conjunction with STK Search and defines a `std::pair` of a bounding box type from STK Search (usually something like `stk::search::Sphere<double>`) to define the location of your target interpolation coordinates, and your `EntityProc` type.
- `Point`: This type is not directly used by the transfer library, but it is used to define other types that are. This type must define a STK Search-compatible point that is used to identify a single set of coordinates for the precise target interpolation locations.
- `ToPointsContainer`: This type defines a random-access container of `Point` types, and is expected to have an API compatible with `std::vector`. This type is used directly by the core transfer library to manage the list of discrete coordinates that will be interpolated to.
- `ToPointsDistanceContainer`: This type defines a random-access container of scalar distances, and is expected to have an API compatible with `std::vector`. This

type is used directly by the core transfer library to track how far each source mesh entity is from the destination coordinates, to aid in finding the closest entity.

As with the send-mesh adapter, there is no required signature for the receive-mesh adapter constructor. The following class methods are required for a receive-mesh adapter:

- `void bounding_boxes(std::vector<BoundingBox> & searchRange)`: This method will be called from `ReducedDependencyGeometricTransfer::coarse_search()` to get the full list of bounding boxes that contain each of the discrete coordinates that the field values will be interpolated to. STK Search will be used to match these boxes up with the best source mesh entities on the sending side.
- `void get_to_points_coordinates(const EntityProcVec & toEntityKeys, ToPointsContainer & toPoints)`: This method is required to provide the list of precise coordinates for each mesh entity that will be receiving interpolated data. These coordinates should be contained within the bounding boxes provided above.
- `void update_values()`: This method will be called on the receive-mesh adapter at the end of `ReducedDependencyGeometricTransfer::apply()` to give the mesh adapter a chance to do any cleanup work after receiving the interpolated data, such as possibly updating field values on any shared entities along parallel boundaries. This method may be empty if there is no work to do.

There are additional class methods implemented on the example receive-mesh adapter that are not needed by the core transfer library, but are still likely to be needed by your `INTERPOLATE` class to service requests from the transfer library. These useful class methods are:

- `void set_field_values(const EntityKey & entityKey, const double * recvInterpValues)`: This function copies the interpolated field data that it receives into the mesh.
- `unsigned total_field_size() const`: This method provides the total number of `double` values summed across all fields that are being interpolated, to help with striding through the data that was communicated.

**Listing 8.13 Interpolation class for reduced-dependency geometric transfer example**
**code/stk/stk_doc_tests/stk_transfer/howToUseReducedDependencyGeometricTransfer.cpp**

```
template<typename SendAdapter, typename RecvAdapter>
class Interpolate
{
public:
  using MeshA = SendAdapter;
  using MeshB = RecvAdapter;
  using EntityKeyA = typename MeshA::EntityKey;
  using EntityKeyB = typename MeshB::EntityKey;
  using EntityProcA = typename MeshA::EntityProc;
  using EntityProcB = typename MeshB::EntityProc;
  using EntityProcRelation = std::pair<EntityProcB, EntityProcA>;
  using EntityProcRelationVec = std::vector<EntityProcRelation>;

  void obtain_parametric_coords(
```

```
      const typename MeshA::EntityProcVec & elemsToInterpolateFrom,
      const MeshA & sendAdapter,
      const typename MeshB::ToPointsContainer & pointsToInterpolateTo,
      typename MeshB::ToPointsDistanceContainer & distanceToInterpolationPoints)
  {
    for (unsigned i = 0; i < elemsToInterpolateFrom.size(); ++i) {
      m_parametricCoords.push_back({0, 0, 0});
      distanceToInterpolationPoints.push_back(0.0);
    }
  }

  void mask_parametric_coords(const std::vector<int> & filterMaskFrom, int fromCount)
  {
    for (unsigned i = 0; i < filterMaskFrom.size(); ++i) {
      if (filterMaskFrom[i]) {
        m_maskedParametricCoords.push_back(m_parametricCoords[i]);
      }
    }
  }

  void apply(MeshB * recvAdapter, MeshA * sendAdapter,
             const typename MeshB::EntityProcVec & toEntityKeysMasked,
             const typename MeshA::EntityProcVec & fromEntityKeysMasked,
             const stk::transfer::ReducedDependencyCommData & commData)
  {
    const unsigned totalFieldSize = sendAdapter->total_field_size();
    std::vector<double> sendInterpValues(fromEntityKeysMasked.size() * totalFieldSize);
    std::vector<double> recvInterpValues(toEntityKeysMasked.size() * totalFieldSize);

    interpolate_from_send_mesh(fromEntityKeysMasked, *sendAdapter, sendInterpValues);
    stk::transfer::do_communication(commData, sendInterpValues, recvInterpValues,
                                    totalFieldSize);
    write_to_recv_mesh(recvInterpValues, toEntityKeysMasked, *recvAdapter);
  }

private:
  void interpolate_from_send_mesh(const typename MeshA::EntityProcVec & fromEntityKeysMasked,
                                  const MeshA & sendAdapter,
                                  std::vector<double> & sendInterpValues)
  {
    unsigned offset = 0;
    for (unsigned i = 0; i < fromEntityKeysMasked.size(); ++i) {
      typename MeshA::EntityKey key = fromEntityKeysMasked[i].id();
      sendAdapter.interpolate_fields(m_maskedParametricCoords[i], key,
             &sendInterpValues[offset]);
      offset += sendAdapter.total_field_size();
    }
  }

  void write_to_recv_mesh(const std::vector<double> & recvInterpValues,
                          const typename MeshB::EntityProcVec & toEntityKeysMasked,
                          MeshB & recvAdapter)
  {
    unsigned offset = 0;
    for (unsigned i = 0; i < toEntityKeysMasked.size(); ++i) {
      typename MeshB::EntityKey key = toEntityKeysMasked[i].id();
      recvAdapter.set_field_values(key, &recvInterpValues[offset]);
      offset += recvAdapter.total_field_size();
    }
  }
  std::vector<std::array<double, 3>> m_parametricCoords;
  std::vector<std::array<double, 3>> m_maskedParametricCoords;
};
```

The INTERPOLATE class template parameter for the transfer object will be discussed next, and an example implementation is illustrated in Listing 8.13. The required types that this class must define are as follows:

- `MeshA`: This is the type of the send-mesh adapter, and is used by the core transfer library to directly interrogate types provided by this mesh adapter in order to communicate with it.
- `MeshB`: This is the type of the receive-mesh adapter, and is used by the core transfer library to directly interrogate types provided by this mesh adapter in order to communicate with it. The sending and receiving meshes need not be of the same type.
- `EntityKeyA`: This is the `EntityKey` type used by the send-mesh adapter, which is an integral type intended to be used as a unique global identifier for mesh entities.
- `EntityKeyB`: This is the `EntityKey` type used by the receive-mesh adapter, which is an integral type intended to be used as a unique global identifier for mesh entities. The two `EntityKey` types need not be the same.
- `EntityProcA`: This is the customized `stk::search::IdentProc` type defined by your send-mesh adapter, and is used to bundle together your unique global identifier for a mesh entity and an MPI processor rank.
- `EntityProcB`: This is the customized `stk::search::IdentProc` type defined by your receive-mesh adapter, and is used to bundle together your unique global identifier for a mesh entity and an MPI processor rank.
- `EntityProcRelation`: This is a pair of `EntityProcA` and `EntityProcB` types in a container that is equivalent to `std::pair`, and is used to store relationships between a unique pair of mesh entities on the sending and receiving meshes.
- `EntityProcRelationVec`: This is a random-access container of `EntityProcRelation` types, equivalent to a `std::vector`.

The `INTERPOLATE` class must be default-constructible, as this is how an instance of it is created by the `ReducedDependencyGeometricTransfer` class. Below are listed all of the methods that this class must provide:

- `void obtain_parametric_coords(const typename MeshA::EntityProcVec & elemsToInterpolateFrom, const MeshA & sendAdapter, const typename MeshB::ToPointsContainer & pointsToInterpolateTo, typename MeshB::ToPointsDistanceContainer & distanceToInterpolationPoints)`: During the coarse search stage of the initial setup, STK Search will be used internally to find candidate mesh entities on the sending side that are near the target coordinates on the receiving side. This function provides information that will be used by the internal fine search to select the single best mesh entity from all of the candidates to provide the final interpolated result. Here, you are given two synchronized lists of candidate mesh entities and the spatial coordinate that they may be asked to provide interpolated values for. You must then fill a list with some meaningful measure of the distance from the mesh entity to the target coordinate. Something like a parametric distance from the centroid of the mesh entity would be a reasonable choice. You also must internally store the information needed to perform the desired interpolation from each entity, such as parametric coordinates if you are performing shape function interpolation. Note that this simple example does not do any actual interpolation, and just stores a zero parametric coordinate and returns a zero distance to the interpolation point. All of the candidate entities will, therefore, be equal in interpolation quality and one of the possible candidates will be selected arbitrarily.

- `void mask_parametric_coords(const std::vector<int> & filterMaskFrom, int fromCount)`: Once the `ReducedDependencyGeometricTransfer` library selects the best mesh entities to provide each interpolated result, this function will be called to inform your `INTERPOLATE` class of the selection. A vector of integers will be provided, one for each of the previously-stored parametric coordinates. A zero value indicates that the entity was not selected and a one value indicates that it was selected. You must now store a shortened list of just the parametric coordinates for the selected mesh entities, to be used in the actual interpolation operation. In the terminology of the transfer library, these are masked entities. The count provided to this function is the total number of selected entities, if it helps with sizing of the storage array.
- `void apply(MeshB * /*recvAdapter*/, MeshA * sendAdapter, const typename MeshB::EntityProcVec & /*toEntityKeysMasked*/, const typename MeshA::EntityProcVec & fromEntityKeysMasked, const stk::transfer::ReducedDependencyCommData & commData)`: This is the main function required by an interpolation class, and it performs the actual interpolation operations and data movement at run-time. This function is given the shortened list of unique entities that will be interpolated from, and clients are responsible for performing the interpolation from all desired fields for each of these mesh entities. Here, the `interpolate_from_send_mesh()` helper function is used to perform the actual interpolation at the previously-stored parametric coordinates for each mesh entity. The results for all desired fields are concatenated and placed in a `std::vector<double>`, and then sent to the appropriate MPI ranks through a call to `stk::transfer::do_communication()`. After the communication, the `write_to_recv_mesh()` function copies the final interpolated results into the receiving mesh. The `totalFieldSize` argument must be the total number of `double` values, aggregated across all interpolated fields, and is required in order to process and communicate the concatenated data values correctly.

### 8.3.2. *Example MPMD Reduced-Dependency Geometric Transfer*

The extreme generality of this transfer capability, where it can operate between any two applications and any two mesh representations, necessitates that users must write a significant amount of code to adapt the workflow to their specific needs. What follows is an example of a highly-simplified transfer of two different data fields of different lengths between two instances of STK Mesh. Again, the mesh database need not be the same on both sides of the transfer and the usage of STK Mesh is not required, although it is convenient for this demonstration.

Listing 8.14 shows the main application. It is formally an SPMD application, although it is set up similarly to an MPMD application using STK Coupling (Chapter 6) to split the MPI communicator in half based on two "colors" for the sending and receiving sides of the transfer. Color 0 is used as the sending side and color 1 is used as the receiving side, and each color may have any number of processors with there being no requirement that the processor counts match. There is no processor overlap between the two sides of the transfer to emulate an MPMD application.

**Listing 8.14 Main application for MPMD reduced-dependency geometric transfer example**
**code/stk/stk_doc_tests/stk_transfer/howToUseReducedDependencyGeometricTransfer.cpp**

```
template <typename INTERPOLATE>
using RDGeomTransfer = stk::transfer::ReducedDependencyGeometricTransfer<INTERPOLATE>;

using SendTransferType = RDGeomTransfer<SendInterpolate<StkSendAdapter, RemoteRecvAdapter>>;
using RecvTransferType = RDGeomTransfer<RecvInterpolate<RemoteSendAdapter, StkRecvAdapter>>;

std::shared_ptr<SendTransferType> setup_send_transfer(MPI_Comm globalComm, BulkDataPtr & bulk,
                                                      const FieldConfig & fieldConfig)
{
  auto sendAdapter = std::make_shared<StkSendAdapter>(globalComm, bulk, "block_1",
                                                      fieldConfig);
  std::shared_ptr<RemoteRecvAdapter> nullRecvAdapter;

  auto sendTransfer = std::make_shared<SendTransferType>(sendAdapter, nullRecvAdapter,
                                                         "SendTransfer", globalComm);

  sendTransfer->initialize();

  return sendTransfer;
}

std::shared_ptr<RecvTransferType> setup_recv_transfer(MPI_Comm globalComm,
                                                      BulkDataPtr & mesh,
                                                      const FieldConfig & fieldConfig)
{
  std::shared_ptr<RemoteSendAdapter> nullSendAdapter;
  auto recvAdapter = std::make_shared<StkRecvAdapter>(globalComm, mesh, "block_1",
                                                      fieldConfig);

  auto recvTransfer = std::make_shared<RecvTransferType>(nullSendAdapter, recvAdapter,
                                                         "RecvTransfer", globalComm);
  recvTransfer->initialize();

  return recvTransfer;
}

TEST(StkTransferHowTo, useReducedDependencyGeometricTransferMPMD)
{
  MPI_Comm commWorld = MPI_COMM_WORLD;
  int numProcs = stk::parallel_machine_size(commWorld);
  int myRank = stk::parallel_machine_rank(commWorld);
  if (numProcs < 2) return;

  int color = myRank < numProcs/2 ? 0 : 1;

  stk::coupling::SplitComms splitComms(commWorld, color);
  splitComms.set_free_comms_in_destructor(true);
  MPI_Comm myComm = splitComms.get_split_comm();

  FieldConfig sendFieldConfig {{"temperature", stk::topology::NODE_RANK, {300.0}},
                               {"velocity", stk::topology::NODE_RANK, {1.0, 2.0, 3.0}}};
  FieldConfig recvFieldConfig {{"temperature", stk::topology::NODE_RANK, {0.0}},
                               {"velocity", stk::topology::NODE_RANK, {0.0, 0.0, 0.0}}};

  if (color == 0) {
```

```
    BulkDataPtr bulk = read_mesh(myComm, "generated:1x1x4", sendFieldConfig);
    auto sendTransfer = setup_send_transfer(commWorld, bulk, sendFieldConfig);
    sendTransfer->apply();
  }
  else if (color == 1) {
    BulkDataPtr bulk = read_mesh(myComm, "generated:1x1x4", recvFieldConfig);
    auto recvTransfer = setup_recv_transfer(commWorld, bulk, recvFieldConfig);
    recvTransfer->apply();
    EXPECT_TRUE(all_field_values_equal(bulk, sendFieldConfig));
  }
}
```

Two nodal fields are configured on both meshes – a scalar `temperature` field and a vector `velocity` field. The fields are given non-zero initial values on the sending side and zero initial values on the receiving side, so that we can easily detect a change once the transfer is complete. Both sides of the transfer must agree on the list of fields so that they can properly encode/decode the serialized MPI data stream that packs all of the interpolated values together.

Both the sending and receiving sides then read the mesh and initialize the coordinate field and the fields that will be transferred. This takes place in the `read_mesh()` function shown previously in Listing 8.10. For this example the meshes are identical, although they are not required to be. The only requirement is that the spatial coordinates have some commonality between the two meshes so that matching locations can be identified between the sending and receiving sides.

For this example, both sides of the transfer then build and configure instances of the corresponding send-transfer and receive-transfer objects that would normally live in the separate sending and receiving applications, respectively. This configuration occurs in the `setup_send_transfer()` and `setup_recv_transfer()` functions, also shown in Listing 8.14. These transfer objects are instances of `stk::transfer::ReducedDependencyGeometricTransfer<INTERPOLATE>` that are templated on a user-provided class that adheres to a specific interface, customized for either send-transfer or receive-transfer interpolation operations. The `INTERPOLATE` class itself may be templated on both a send-mesh adapter and a receive-mesh adapter class despite only one being used in each context, for the purposes of interface uniformity.

The `setup_send_transfer()` function illustrated in Listing 8.14 constructs the sending transfer object templated on the example `SendInterpolate<StkSendAdapter, RemoteRecvAdapter>` class, while the `setup_recv_transfer()` function constructs the receiving transfer object templated on the example `RecvInterpolate<RemoteSendAdapter, StkRecvAdapter>` class. In both cases, a dummy mesh adapter is used for the other remote side. It does not need any kind of dependence on the actual mesh representation that will be communicated with, which is one of the central benefits of the `ReducedDependencyGeometricTransfer`. This transfer capability can be hooked together at run-time with any other remote application that implements the required transfer API. Because of this, there are a few rules that must be followed to ensure compatibility, such as the sizes of various defined types. These will be discussed in more detail below.

The `stk::transfer::ReducedDependencyGeometricTransfer` class has constructor arguments of a `std::shared_ptr` to instances of both the send-mesh adapter and the receive-mesh adapter, while the `INTERPOLATE` class is default-constructed internally. If this is the sending side of an MPMD transfer, then it is expected that the remote receive-mesh adapter instance will be null and if this is the receiving side of an MPMD transfer, then it is expected that the remote send-mesh adapter instance will be null. These remote mesh adapters will not be used at run-time if null, which allows them to have no real functionality other than simply providing some generic type definitions.

Once the sending and receiving transfer objects are constructed, they are configured by making a call to their `initialize()` method. This is a shorthand for making sequential calls to `coarse_search()` and `communication()` for the different stages of initial setup. A call is also made to `local_search()`, but this class method does nothing for this transfer capability. The `coarse_search()` method internally uses STK Search (Chapter 7) to identify candidate mesh entities (elements, faces, etc.) on the sending side that correspond to the target coordinates on the receiving side. The `communication()` method then shares the lists of mesh entities among the processors and identifies an optimal set of unique one-to-one mappings between the two meshes. User-provided supporting functions for each of these initialization calls will be discussed in the mesh adapter and `INTERPOLATE` class descriptions below.

This initial configuration work only needs to be done once if the meshes are static. If either mesh is modified or if entities in either mesh deform and change their coordinates, then this search and communication work will need to be re-done in a parallel-consistent manner across both applications before the actual transfer operation occurs.

Once the transfer objects have been constructed and configured, the applications may trigger a data transfer at any time by simultaneously calling `apply()` on the transfer objects, as shown in Listing 8.14. This will do the actual interpolation on the sending side, package it up, and send it over to the receiving side where it can be inserted into the destination mesh.

This demonstration application has a final call to `all_field_values_equal()` on the receiving side to ensure that the sent values get written to the receiving mesh correctly.

For this example, the send-mesh adapter used on the sending side is identical to what was already shown for the SPMD example in Listing 8.11. The dummy remote receive-mesh adapter is shown in Listing 8.15. As described previously, it has no real functionality other than satisfying dependencies of the transfer library and defining a few generic types. Compile-time errors will results if the requirements are not fulfilled, although the main requirement is that the `EntityKey` is an 8-byte integral type.

**Listing 8.15 Remote Receive-Mesh Adapter class for reduced-dependency geometric transfer example code/stk/stk_doc_tests/stk_transfer/howToUseReducedDependencyGeometricTransfer.cpp**

```
class RemoteRecvAdapter
{
public:
  using EntityKey = uint64_t;
  using EntityProc = stk::search::IdentProc<EntityKey>;
  using EntityProcVec = std::vector<EntityProc>;
  using BoundingBox = std::pair<stk::search::Sphere<double>, EntityProc>;

  using Point = stk::search::Point<double>;
  using ToPointsContainer = std::vector<Point>;
  using ToPointsDistance = double;
  using ToPointsDistanceContainer = std::vector<ToPointsDistance>;

  void bounding_boxes(std::vector<BoundingBox> & ) const {}
```

```
  void get_to_points_coordinates(const EntityProcVec & , ToPointsContainer & ) {}
  void update_values() {}
};
```

The receive-mesh adapter used on the receiving side of the transfer is also identical to what was already shown previously in Listing 8.12. The dummy remote send-mesh adapter is shown in Listing 8.16 and has similar restrictions to the dummy remote receive-mesh adapter.

**Listing 8.16 Remote Send-Mesh Adapter class for reduced-dependency geometric transfer example**
**code/stk/stk_doc_tests/stk_transfer/howToUseReducedDependencyGeometricTransfer.cpp**

```
class RemoteSendAdapter
{
public:
  using EntityKey = uint64_t;
  using EntityProc = stk::search::IdentProc<EntityKey>;
  using EntityProcVec = std::vector<EntityProc>;
  using BoundingBox = std::pair<stk::search::Box<double>, EntityProc>;

  void bounding_boxes(std::vector<BoundingBox> & ) const {}
  void update_values() {}
};
```

The two INTERPOLATE classes are also similar to that for the SPMD example, except that the required functionality is subdivided to one or the other version.

**Listing 8.17 Send-Interpolate class for MPMD reduced-dependency geometric transfer example**
**code/stk/stk_doc_tests/stk_transfer/howToUseReducedDependencyGeometricTransfer.cpp**

```
template<typename SendAdapter, typename RecvAdapter>
class SendInterpolate
{
public:
  using MeshA = SendAdapter;
  using MeshB = RecvAdapter;
  using EntityKeyA = typename MeshA::EntityKey;
  using EntityKeyB = typename MeshB::EntityKey;
  using EntityProcA = typename MeshA::EntityProc;
  using EntityProcB = typename MeshB::EntityProc;
  using EntityProcRelation = std::pair<EntityProcB, EntityProcA>;
  using EntityProcRelationVec = std::vector<EntityProcRelation>;

  void obtain_parametric_coords(
      const typename MeshA::EntityProcVec & elemsToInterpolateFrom,
      const MeshA & sendAdapter,
      const typename MeshB::ToPointsContainer & pointsToInterpolateTo,
      typename MeshB::ToPointsDistanceContainer & distanceToInterpolationPoints)
  {
    for (unsigned i = 0; i < elemsToInterpolateFrom.size(); ++i) {
      m_parametricCoords.push_back({0, 0, 0});
      distanceToInterpolationPoints.push_back(0.0);
    }
  }

  void mask_parametric_coords(const std::vector<int> & filterMaskFrom, int fromCount)
  {
    for (unsigned i = 0; i < filterMaskFrom.size(); ++i) {
      if (filterMaskFrom[i]) {
        m_maskedParametricCoords.push_back(m_parametricCoords[i]);
      }
    }
  }
```

```
  void apply(MeshB * /*recvAdapter*/, MeshA * sendAdapter,
             const typename MeshB::EntityProcVec & /*toEntityKeysMasked*/,
             const typename MeshA::EntityProcVec & fromEntityKeysMasked,
             const stk::transfer::ReducedDependencyCommData & commData)
  {
    const unsigned totalFieldSize = sendAdapter->total_field_size();
    std::vector<double> sendInterpValues(fromEntityKeysMasked.size() * totalFieldSize);
    interpolate_from_send_mesh(fromEntityKeysMasked, *sendAdapter, sendInterpValues);

    std::vector<double> recvInterpValues;  // Unused
    stk::transfer::do_communication(commData, sendInterpValues, recvInterpValues,
                                    totalFieldSize);
  }

private:
  void interpolate_from_send_mesh(const typename MeshA::EntityProcVec & fromEntityKeysMasked,
                                  const MeshA & sendAdapter,
                                  std::vector<double> & sendInterpValues)
  {
    unsigned offset = 0;
    for (unsigned i = 0; i < fromEntityKeysMasked.size(); ++i) {
      typename MeshA::EntityKey key = fromEntityKeysMasked[i].id();
      sendAdapter.interpolate_fields(m_maskedParametricCoords[i], key,
             &sendInterpValues[offset]);
      offset += sendAdapter.total_field_size();
    }
  }

  std::vector<std::array<double, 3>> m_parametricCoords;
  std::vector<std::array<double, 3>> m_maskedParametricCoords;
};
```

An example `SendInterpolate` class is shown in Listing 8.17. As with the mesh adapter classes, a number of types must be defined to satisfy the `ReducedDependencyGeometricTransfer` library. These types are as follows:

- `MeshA`: This is the type of the send-mesh adapter, and is used by the core transfer library to directly interrogate types provided by this mesh adapter in order to communicate with it.
- `MeshB`: This is the type of the receive-mesh adapter, and is only used to define further types required for the code to compile. The receive mesh adapter instance is expected to be null when passed as a constructor argument while building the transfer object on the sending side. Note that the `A`/`B` nomenclature is used in these classes to refer to the sending-side and receiving-side versions of a type, respectively.
- `EntityKeyA`: This is the `EntityKey` type used by the send-mesh adapter, which is an integral type intended to be used as a unique global identifier for mesh entities.
- `EntityKeyB`: This is the `EntityKey` type used by the receive-mesh adapter, which is an integral type intended to be used as a unique global identifier for mesh entities. Again, this type is unused on the sending side.
- `EntityProcA`: This is the customized `stk::search::IdentProc` type defined by your send-mesh adapter, and is used to bundle together your unique global identifier for a mesh entity and an MPI processor rank.
- `EntityProcB`: This is the customized `stk::search::IdentProc` type defined by your receive-mesh adapter, and is unused on the sending side.

- `EntityProcRelation`: This is a pair of `EntityProcA` and `EntityProcB` types in a container that is equivalent to `std::pair`, and is used to store relationships between a unique pair of mesh entities on the sending and receiving side of the transfer.
- `EntityProcRelationVec`: This is a random-access container of `EntityProcRelation` types, equivalent to a `std::vector`.

The `SendInterpolate` class must be default-constructible, as this is how an instance of it is created by the `ReducedDependencyGeometricTransfer` class. Below are listed the required methods that this class must provide:

- `void obtain_parametric_coords(const typename MeshA::EntityProcVec & elemsToInterpolateFrom, const MeshA & sendAdapter, const typename MeshB::ToPointsContainer & pointsToInterpolateTo, typename MeshB::ToPointsDistanceContainer & distanceToInterpolationPoints)`: During the coarse search stage of the initial setup, STK Search will be used internally to find candidate mesh entities on the sending side that are near the target coordinates on the receiving side. This function provides information that will be used by the internal fine search to select the single best mesh entity from all of the candidates to provide the final interpolated result. Here, you are given two synchronized lists of candidate mesh entities and the spatial coordinate that they may be asked to provide interpolated values for. You must then fill a list with some meaningful measure of the distance from the mesh entity to the target coordinate. Something like a parametric distance from the centroid of the mesh entity would be a reasonable choice. You also must internally store the information needed to perform the desired interpolation from each entity, such as parametric coordinates if you are performing shape function interpolation. Note that this simple example does not do any actual interpolation, and just stores a zero parametric coordinate and returns a zero distance to the interpolation point. All of the candidate entities will, therefore, be equal in interpolation quality and one of the possible candidates will be selected arbitrarily.
- `void mask_parametric_coords(const std::vector<int> & filterMaskFrom, int fromCount)`: Once the `ReducedDependencyGeometricTransfer` library selects the best mesh entities to provide each interpolated result, this function will be called to inform your `INTERPOLATE` class of the selection. A vector of integers will be provided, one for each of the previously-stored parametric coordinates. A zero value indicates that the entity was not selected and a one value indicates that it was selected. You must now store a shortened list of just the parametric coordinates for the selected mesh entities, to be used in the actual interpolation operation. In the terminology of the transfer library, these are masked entities. The count provided to this function is the total number of selected entities, if it helps with sizing of the storage array.
- `void apply(MeshB * /*recvAdapter*/, MeshA * sendAdapter, const typename MeshB::EntityProcVec & /*toEntityKeysMasked*/, const typename MeshA::EntityProcVec`

& fromEntityKeysMasked, const stk::transfer::ReducedDependencyCommData & commData): This is the main function required by an interpolation class, and it performs the actual interpolation operations at run-time and sends the results to the receiving side of the transfer. This function is given the shortened list of unique entities that will be interpolated from, and clients are responsible for performing the interpolation for all desired fields for each of these mesh entities. Here, the `interpolate_from_send_mesh()` helper function is used to perform the actual interpolation at the previously-stored parametric coordinates for each mesh entity. The results for all desired fields are concatenated and placed in a `std::vector<double>`, and then sent to the other application through a call to `stk::transfer::do_communication()`. The `totalFieldSize` argument must be the total number of `double` values, aggregated across all interpolated fields, and is required in order to process the data for each mesh entity with a proper stride.

**Listing 8.18 Receive-Interpolate class for MPMD reduced-dependency geometric transfer example code/stk/stk_doc_tests/stk_transfer/howToUseReducedDependencyGeometricTransfer.cpp**

```cpp
template<typename SendAdapter, typename RecvAdapter>
class RecvInterpolate
{
public:
  using MeshA = SendAdapter;
  using MeshB = RecvAdapter;
  using EntityKeyA = typename MeshA::EntityKey;
  using EntityKeyB = typename MeshB::EntityKey;
  using EntityProcA = typename MeshA::EntityProc;
  using EntityProcB = typename MeshB::EntityProc;
  using EntityProcRelation = std::pair<EntityProcB, EntityProcA>;
  using EntityProcRelationVec = std::vector<EntityProcRelation>;

  void obtain_parametric_coords(const typename MeshA::EntityProcVec , MeshA & ,
                                const typename MeshB::ToPointsContainer & ,
                                const typename MeshB::ToPointsDistanceContainer & ) {}

  void mask_parametric_coords(const std::vector<int> & , int ) {}

  void apply(MeshB * recvAdapter, MeshA * /*sendAdapter*/,
             const typename MeshB::EntityProcVec & toEntityKeysMasked,
             const typename MeshA::EntityProcVec & /*fromEntityKeysMasked*/,
             const stk::transfer::ReducedDependencyCommData & comm_data)
  {
    const unsigned totalFieldSize = recvAdapter->total_field_size();
    std::vector<double> sendInterpValues;  // Unused
    std::vector<double> recvInterpValues(toEntityKeysMasked.size() * totalFieldSize);
    stk::transfer::do_communication(comm_data, sendInterpValues, recvInterpValues,
                                    totalFieldSize);

    write_to_recv_mesh(recvInterpValues, toEntityKeysMasked, *recvAdapter);
  }

private:
  void write_to_recv_mesh(const std::vector<double> & recvInterpValues,
                          const typename MeshB::EntityProcVec & toEntityKeysMasked,
                          MeshB & recvAdapter)
  {
    unsigned offset = 0;
    for (unsigned i = 0; i < toEntityKeysMasked.size(); ++i) {
      typename MeshB::EntityKey key = toEntityKeysMasked[i].id();
      recvAdapter.set_field_values(key, &recvInterpValues[offset]);
      offset += recvAdapter.total_field_size();
    }
```

```
    }

};
```

The RecvInterpolate class has the same signature requirements as the SendInterpolate class since this is the only template parameter of the ReducedDependencyGeometricTransfer class. A simplified example of an INTERPOLATE class for receiving transfer data is shown in Listing 8.18. Below are listed the required types to be provided by the receive interpolation class:

- MeshA: This is the type of the send-mesh adapter template parameter, and is only used to define further types required for the code to compile. The send-mesh adapter instance is expected to be null when passed as a constructor argument while building the transfer object on the receiving side.
- MeshB: This is the type of the receive-mesh adapter template parameter, and is used by the core transfer library to directly interrogate types provided by this mesh adapter in order to communicate with it. Note that the A/B nomenclature is used by the transfer library to refer to the sending-side and receiving-side versions of a type, respectively.
- EntityKeyA: This is the EntityKey type used by the sending-mesh adapter, which is an integral type intended to be used as a unique global identifier for mesh entities. Again, this type is unused on the receiving side.
- EntityKeyB: This is the EntityKey type used by the receiving mesh adapter, which is an integral type intended to be used as a unique global identifier for mesh entities.
- EntityProcA: This is the customized stk::search::IdentProc type defined by your send-mesh adapter, and is unused on the receiving side.
- EntityProcB: This is the customized stk::search::IdentProc type defined by your receive-mesh adapter, and is used to bundle together your unique global identifier for a mesh entity and an MPI processor rank.
- EntityProcRelation: This is a pair of EntityProcA and EntityProcB types in a container that is equivalent to std::pair, and is used to store relationships between a unique pair of mesh entities on the sending and receiving side of the transfer.
- EntityProcRelationVec: This is a random-access container of EntityProcRelation types, equivalent to a std::vector.

The required interpolation class methods perform different tasks on the receiving side than on the sending side, despite having the same signature. The behaviors required to be implemented by the RecvInterpolate class are as follows:

- **void** obtain_parametric_coords(**const typename** MeshA::EntityProcVec & elemsToInterpolateFrom, **const** MeshA & sendAdapter, **const typename** MeshB::ToPointsContainer & pointsToInterpolateTo, **typename** MeshB::ToPointsDistanceContainer & distanceToInterpolationPoints): This class method is unused on the receiving side, although it is still required to be implemented to satisfy the signature expectations of the `ReducedDependencyGeometricTransfer` class. Its implementation should be empty.

- `void mask_parametric_coords(const std::vector<int> & filterMaskFrom, int fromCount)`: This class method is also unused on the receiving side, and should be empty.
- `void apply(MeshB * /*recvAdapter*/, MeshA * sendAdapter, const typename MeshB::EntityProcVec & /*toEntityKeysMasked*/, const typename MeshA::EntityProcVec & fromEntityKeysMasked, const stk::transfer::ReducedDependencyCommData & commData)`: This is the main function required by an interpolation class, and it manages the actual interpolated data movement at run-time. This function is given the shortened list of unique entities that will receive the interpolated data, although this list is identical to the original list because the destination coordinates for interpolation are treated as immutable. In this method, the user is required to first perform the communication with the other application by calling the `stk::transfer::do_communication()` function, which receives the final list of interpolated data for each interpolation point. The user then stores the interpolated results in the target field data inside its mesh adapter. One data point for each field will be aggregated into a contiguous block of data for each mesh entity, synchronized with the list of entities provided to this function. This length can be viewed as the stride between adjacent entity's data, and is provided as an argument to the communication routine.

This page intentionally left blank.

# 9. STK BALANCE

The STK Balance module provides load balancing capabilities for which many options are configurable by the application teams. STK Balance interfaces with Zoltan2 (need reference) which provides geometric and graph based decomposition capabilities. STK Balance is scalable and able to balance very large (billions of elements) meshes.

## 9.1. Stand-alone Decomposition Tool

There is a stand-alone executable called `stk_balance` which can be used to decompose a mesh using a graph-based algorithm. Please see `stk_balance --help` for current usage information. This tool must be run on the number of processors desired for the decomposition, as follows:

```
mpirun --np 256 stk_balance input_mesh.g output_dir
```

where `input_mesh.g` is the serial Exodus mesh file to decompose and `output_dir` is the optional output directory where the decomposed files are to be written rather than the current directory.

The default behavior of `stk_balance` is to perform a graph-based decomposition using Parmetis as the partitioning tool. A proximity search will be used to group entities on the same processor that may be in contact. A default absolute search tolerance of 0.0001 is used for faces and a tolerance of 3 times the radius is used for particles. The optional `--sm` flag changes the search tolerances to be more similar to what the SM applications use for contact search. This includes a face search tolerance of 15% of the second-shortest face edge. Relative graph vertex weights are increased and graph edge weights are decreased for entities found during search. The optional `--sd` flag uses the default search tolerances but adds an algorithm to handle “spider” elements. The extreme connectivities found when using spider elements can confuse the partitioner, so this algorithm modification performs an initial decomposition using only the volume elements, and then moves the beam elements in each spider to the same owner as the exposed volume element mesh nodes that they are connected to.

## 9.2. Geometric Balancing

The following tests show the basic usage of the STK Balance with the RCB (Recursive Coordinate Bisection - need reference) method.

**Listing 9.1 Stk Balance RCB Example**
**code/stk/stk_doc_tests/stk_balance/howToUseStkBalance.cpp**

```
TEST_F(StkBalanceHowTo, UseRebalanceWithGeometricMethods)
{
    if(stk::parallel_machine_size(get_comm()) == 2)
    {
        setup_mesh("generated:4x4x4|sideset:xX", stk::mesh::BulkData::NO_AUTO_AURA);

        RcbSettings balanceSettings;
        stk::balance::balanceStkMesh(balanceSettings, get_bulk());

        EXPECT_TRUE(is_mesh_balanced(get_bulk()));
    }
}
```

**Listing 9.2 Stk Balance Settings For RCB**
**code/stk/stk_doc_tests/stk_balance/howToUseStkBalance.cpp**

```
class RcbSettings : public stk::balance::BalanceSettings
{
public:
    RcbSettings() {}
    virtual ~RcbSettings() {}

    virtual bool isIncrementalRebalance() const { return false; }
    virtual std::string getDecompMethod() const { return std::string("rcb"); }
    virtual std::string getCoordinateFieldName() const { return std::string("coordinates"); }
    virtual bool shouldPrintMetrics() const { return true; }
};
```

## 9.3. Graph Based Balancing With Parmetis

The following tests show the basic usage of the STK Balance with Parmetis (need reference - graph based decomposition). This allows the application developer to set vertex and edge weights of the graph. In addition, it provides the flexibility to change what defines an edge between two vertices. In this context, a vertex is an element, and an edge is a connection between elements.

**Listing 9.3 Stk Balance API Parmetis Example**
**code/stk/stk_doc_tests/stk_balance/howToUseStkBalance.cpp**

```
TEST_F(StkBalanceHowTo, UseRebalanceWithParmetis)
{
    if(stk::parallel_machine_size(get_comm()) == 2)
    {
        setup_mesh("generated:4x4x4|sideset:xX", stk::mesh::BulkData::NO_AUTO_AURA);

        ParmetisSettings balanceSettings;
        stk::balance::balanceStkMesh(balanceSettings, get_bulk());

        EXPECT_TRUE(is_mesh_balanced(get_bulk()));
    }
}
```

**Listing 9.4 Stk Balance Settings For Parmetis**
**code/stk/stk_doc_tests/stk_balance/howToUseStkBalance.cpp**

```cpp
class ParmetisSettings : public stk::balance::GraphCreationSettings
{
public:
    virtual std::string getDecompMethod() const { return "parmetis"; }

    size_t getNumNodesRequiredForConnection(stk::topology element1Topology, stk::topology
            element2Topology) const
    {
        const int noConnection = 1000;
        const int s = noConnection;
        const static int connectionTable[7][7] = {
            {1, 1, 1, 1, 1, 1, s}, // 0 dim
            {1, 1, 1, 1, 1, 1, s}, // 1 dim
            {1, 1, 2, 3, 2, 3, s}, // 2 dim linear
            {1, 1, 3, 3, 3, 3, s}, // 3 dim linear
            {1, 1, 2, 3, 3, 4, s}, // 2 dim higher-order
            {1, 1, 3, 3, 4, 4, s}, // 3 dim higher-order
            {s, s, s, s, s, s, s}  // super element
        };

        int element1Index = getConnectionTableIndex(element1Topology);
        int element2Index = getConnectionTableIndex(element2Topology);

        return connectionTable[element1Index][element2Index];
    }

    virtual double getGraphEdgeWeight(stk::topology element1Topology, stk::topology
            element2Topology) const
    {
        const double noConnection = 0;
        const double s = noConnection;
        const double largeWeight = 1000;
        const double L = largeWeight;
        const double twoDimWeight = 5;
        const double q = twoDimWeight;
        const double defaultWeight = 1.0;
        const double D = defaultWeight;
        const static double weightTable[7][7] = {
            {L, L, L, L, L, L, s}, // 0 dim
            {L, L, L, L, L, L, s}, // 1 dim
            {L, L, q, q, q, q, s}, // 2 dim linear
            {L, L, q, D, q, D, s}, // 3 dim linear
            {L, L, q, q, q, q, s}, // 2 dim higher-order
            {L, L, q, D, q, D, s}, // 3 dim higher-order
            {s, s, s, s, s, s, s}  // super element
        };

        int element1Index = getConnectionTableIndex(element1Topology);
        int element2Index = getConnectionTableIndex(element2Topology);

        return weightTable[element1Index][element2Index];
    }

    using BalanceSettings::getGraphVertexWeight;

    virtual int getGraphVertexWeight(stk::topology type) const
    {
        switch(type)
        {
            case stk::topology::PARTICLE:
            case stk::topology::LINE_2:
            case stk::topology::BEAM_2:
                return 1;
            case stk::topology::SHELL_TRIANGLE_3:
                return 3;
            case stk::topology::SHELL_TRIANGLE_6:
                return 6;
            case stk::topology::SHELL_QUADRILATERAL_4:
```

```
                return 6;
            case stk::topology::SHELL_QUADRILATERAL_8:
                return 12;
            case stk::topology::HEXAHEDRON_8:
                return 3;
            case stk::topology::HEXAHEDRON_20:
                return 12;
            case stk::topology::TETRAHEDRON_4:
                return 1;
            case stk::topology::TETRAHEDRON_10:
                return 3;
            case stk::topology::WEDGE_6:
                return 2;
            case stk::topology::WEDGE_15:
                return 12;
            default:
                if ( type.is_superelement( ))
                {
                    return 10;
                }
                throw("Invalid Element Type In WeightsOfElement");
        }
    }
};
```

## 9.4. Graph Based Balancing With Parmetis Using Search

The following tests show the basic usage of the STK Balance with Parmetis (need reference - graph based decomposition) where a coarse search is used to insert edges into the graph. The search settings will override the vertex weights of the graph if defined on the settings.

**Listing 9.5 Stk Balance API Parmetis With Search Example code/stk/stk_doc_tests/stk_balance/howToUseStkBalance.cpp**

```
TEST_F(StkBalanceHowTo, UseRebalanceWithParmetisAugmentedWithSearch)
{
    if(stk::parallel_machine_size(get_comm()) == 2)
    {
        setup_mesh("generated:4x4x4|sideset:xX", stk::mesh::BulkData::NO_AUTO_AURA);

        ParmetisWithSearchSettings balanceSettings;
        stk::balance::balanceStkMesh(balanceSettings, get_bulk());

        EXPECT_TRUE(is_mesh_balanced(get_bulk()));
    }
}
```

**Listing 9.6 Stk Balance Settings For Parmetis With Search code/stk/stk_doc_tests/stk_balance/howToUseStkBalance.cpp**

```
class ParmetisWithSearchSettings : public ParmetisSettings
{
    using ParmetisSettings::getToleranceForFaceSearch;
    virtual bool includeSearchResultsInGraph() const { return true; }
    virtual double getToleranceForFaceSearch() const { return 0.0001; }
    virtual double getVertexWeightMultiplierForVertexInSearch() const { return 6.0; }
    virtual double getGraphEdgeWeightForSearch() const { return 1000; }
};
```

## 9.5. Graph Based Balancing Using A Field For Vertex Weights

The following tests show the basic usage of the STK Balance where an application specified field is used to set vertex weights.

**Listing 9.7 Stk Balance API Using A Field To Set Vertex Weights Example**
**code/stk/stk_doc_tests/stk_balance/howToUseStkBalance.cpp**

```
TEST_F(StkBalanceHowTo, UseRebalanceWithFieldSpecifiedVertexWeights)
{
    if(stk::parallel_machine_size(get_comm()) == 2)
    {
        setup_empty_mesh(stk::mesh::BulkData::NO_AUTO_AURA);
        stk::mesh::Field<double> &weightField =
             get_meta().declare_field<double>(stk::topology::ELEM_RANK, "vertex_weights");
        stk::mesh::put_field_on_mesh(weightField, get_meta().universal_part(), nullptr);
        stk::io::fill_mesh("generated:4x4x4|sideset:xX", get_bulk());
        set_vertex_weights(get_bulk(), get_meta().locally_owned_part(), weightField);

        FieldVertexWeightSettings balanceSettings(weightField);
        stk::balance::balanceStkMesh(balanceSettings, get_bulk());

        EXPECT_TRUE(is_mesh_balanced_wrt_weight(get_bulk(), weightField));
    }
}
```

**Listing 9.8 Stk Balance Settings For Setting Vertex Weights Using A Field**
**code/stk/stk_doc_tests/stk_balance/howToUseStkBalance.cpp**

```
class FieldVertexWeightSettings : public stk::balance::GraphCreationSettings
{
public:
    FieldVertexWeightSettings(const stk::balance::DoubleFieldType &weightField,
                              const double defaultWeight = 0.0)
  {
    setVertexWeightMethod(stk::balance::VertexWeightMethod::FIELD);
    setVertexWeightFieldName(weightField.name());
    setDefaultFieldWeight(defaultWeight);
  }

    virtual ~FieldVertexWeightSettings() = default;

    virtual double getGraphEdgeWeight(stk::topology element1Topology, stk::topology
             element2Topology) const { return 1.0; }

    virtual int getGraphVertexWeight(stk::topology type) const { return 1; }
    virtual double getImbalanceTolerance() const { return 1.0001; }
    virtual std::string getDecompMethod() const { return "rcb"; }

protected:
    FieldVertexWeightSettings() = delete;
    FieldVertexWeightSettings(const FieldVertexWeightSettings&) = delete;
    FieldVertexWeightSettings& operator=(const FieldVertexWeightSettings&) = delete;
};
```

## 9.6. STK Balancing Using Multiple Criteria

The following tests show the usage of the STK Balance when balancing different grouping of entities at the same time, e.g., a multi-physics balancing. Currently, multi-criteria rebalancing is related to balancing a mesh using multiple selectors or fields or both. The next two sections show the API for selector and field based multi-criteria balancing.

### 9.6.1. Multiple Criteria Related To Selectors

This shows the API for using multiple selectors to balance a mesh, e.g., a multi-physics mesh.

**Listing 9.9 Stk Balance API Using Selectors To Balance A Mesh Example**
**code/stk/stk_doc_tests/stk_balance/howToUseStkBalance.cpp**

```
TEST_F(StkBalanceHowTo, UseRebalanceWithMultipleCriteriaWithSelectors)
{
    if(stk::parallel_machine_size(get_comm()) == 2)
    {
        setup_empty_mesh(stk::mesh::BulkData::NO_AUTO_AURA);
        stk::mesh::Part &part1 = get_meta().declare_part("madeup_part_1",
            stk::topology::ELEM_RANK);
        stk::mesh::Part &part2 = get_meta().declare_part("part_2", stk::topology::ELEM_RANK);
        stk::io::fill_mesh("generated:4x4x4|sideset:xX", get_bulk());

        put_elements_in_different_parts(get_bulk(), part1, part2);

        std::vector<stk::mesh::Selector> selectors = { part1, part2 };

        MultipleCriteriaSelectorSettings balanceSettings;
        stk::balance::balanceStkMesh(balanceSettings, get_bulk(), selectors);

        verify_mesh_balanced_wrt_selectors(get_bulk(), selectors);
    }
}
```

**Listing 9.10 Stk Balance Settings For Multi-criteria Balancing Using Selectors**
**code/stk/stk_doc_tests/stk_balance/howToUseStkBalance.cpp**

```
class MultipleCriteriaSelectorSettings : public ParmetisSettings
{
public:
    MultipleCriteriaSelectorSettings() { }
    virtual ~MultipleCriteriaSelectorSettings() = default;

    virtual bool isMultiCriteriaRebalance() const { return true;}

protected:
    MultipleCriteriaSelectorSettings(const MultipleCriteriaSelectorSettings&) = delete;
    MultipleCriteriaSelectorSettings& operator=(const MultipleCriteriaSelectorSettings&) =
            delete;
};
```

### 9.6.2. Multiple Criteria Related To Multiple Fields

This shows the API for using multiple fields to balance a mesh, e.g., a multi-physics mesh.

**Listing 9.11 Stk Balance API Using Fields To Balance A Mesh Example**
**code/stk/stk_doc_tests/stk_balance/howToUseStkBalance.cpp**

```
TEST_F(StkBalanceHowTo, UseRebalanceWithMultipleCriteriaWithFields)
{
```

```
    if(stk::parallel_machine_size(get_comm()) == 2)
    {
        setup_empty_mesh(stk::mesh::BulkData::NO_AUTO_AURA);
        stk::mesh::Field<double> &weightField1 =
             get_meta().declare_field<double>(stk::topology::ELEM_RANK, "vertex_weights1");
        stk::mesh::put_field_on_mesh(weightField1, get_meta().universal_part(), nullptr);

        stk::mesh::Field<double> &weightField2 =
             get_meta().declare_field<double>(stk::topology::ELEM_RANK, "vertex_weights2");
        stk::mesh::put_field_on_mesh(weightField2, get_meta().universal_part(), nullptr);

        stk::io::fill_mesh("generated:4x4x4|sideset:xX", get_bulk());

        set_vertex_weights_checkerboard(get_bulk(), get_meta().locally_owned_part(),
             weightField1, weightField2);

        std::vector<stk::mesh::Field<double>*> critFields = { &weightField1, &weightField2 };
        MultipleCriteriaFieldSettings balanceSettings(critFields);
        stk::balance::balanceStkMesh(balanceSettings, get_bulk());

        verify_mesh_balanced_wrt_fields(get_bulk(), critFields);
    }
}
```

**Listing 9.12 Stk Balance Settings For Multi-criteria Balancing Using Fields**
**code/stk/stk_doc_tests/stk_balance/howToUseStkBalance.cpp**

```
class MultipleCriteriaFieldSettings : public ParmetisSettings
{
public:
    MultipleCriteriaFieldSettings(const std::vector<stk::mesh::Field<double>*> critFields,
                                  const double default_weight = 0.0)
    {
      setNumCriteria(critFields.size());
      setVertexWeightMethod(stk::balance::VertexWeightMethod::FIELD);
      for (unsigned i = 0; i < critFields.size(); ++i) {
        setVertexWeightFieldName(critFields[i]->name(), i);
      }
      setDefaultFieldWeight(default_weight);
    }
    virtual ~MultipleCriteriaFieldSettings() override = default;

    virtual bool isMultiCriteriaRebalance() const { return true;}

protected:
    MultipleCriteriaFieldSettings() = delete;
    MultipleCriteriaFieldSettings(const MultipleCriteriaFieldSettings&) = delete;
    MultipleCriteriaFieldSettings& operator=(const MultipleCriteriaFieldSettings&) = delete;
};
```

This page intentionally left blank.

# 10. STK SIMD

The STK *SIMD* module provides a computationally efficient way of performing mathematical operations on vector arrays of double and float types. The key components of this library are

- stk::simd::Doubles
- stk::simd::Floats

These types are actually a packed array of size stk::simd::ndoubles and stk::simd::nfloats, respectively. These vector length sizes can be vary from platform to platform. It is important that the user of the stk_simd library writes their algorithms so that changing ndoubles (or nfloats) does not change behavior. Most basic mathematical operations are implemented to work on these simd types.

## 10.1. Example STK SIMD usage

This test gives an example of how to apply a simple nonlinear operations on all the entries of an array using *SIMD* types, in a way which does not assume a specific vector length. Three essential steps are necessary to accomplish this

- the data from the input array must be loaded into the *SIMD* type
- the mathematical operations are applied to the *SIMD* data and stored temporarily into an output *SIMD* type
- the data is stored into the output array

**Listing 10.1 Example of simple operations using STK SIMD**
**code/stk/stk_doc_tests/stk_simd/simpleStkSimd.cpp**

```
TEST(StkSimdHowTo, simdSimdTest)
{
  const int N = 512; // this is a multiple of the simd width
                     // if this is not true, the remainder
                     // must be handled appropriately

  static_assert( N % stk::simd::ndoubles == 0, "Required to be a multiple of ndoubles");

  std::vector<double, non_std::AlignedAllocator<double,64> > x(N);
  std::vector<double, non_std::AlignedAllocator<double,64> > y(N);
  std::vector<double, non_std::AlignedAllocator<double,64> > solution(N);

  for (int n=0; n < N; ++n) {
    x[n] = (rand()-0.5)/RAND_MAX;
    y[n] = (rand()-0.5)/RAND_MAX;
  }

  for (int n=0; n < N; n+=stk::simd::ndoubles) {
    const stk::simd::Double xl = stk::simd::load(&x[n]);
```

```
    const stk::simd::Double yl = stk::simd::load(&y[n]);
    stk::simd::Double zl = stk::math::abs(xl) * stk::math::exp(yl);
    stk::simd::store(&solution[n],zl);
  }

  const double epsilon = 1.e-14;
  for (int n=0; n < N; ++n) {
    EXPECT_NEAR( std::abs(x[n]) * std::exp(y[n]), solution[n], epsilon );
  }
}
```

# 11. STK MIDDLE MESH

The STK Middle Mesh product creates a surface mesh formed from the intersection of two input surface meshes. This uses a different mesh data structure that supports very fast mesh modification.

## 11.1. Middle Mesh Data Structure

This section describes how to use the `stk::middle_mesh::mesh::Mesh` data structure. This data structure is intended for fast mesh modification. At present, it is only capable of representing surface meshes (ie. 2D elements in 3D space).

The data structure has several features that distinguish it from other mesh data structures, and are important for fast mesh modification:

- All IDs are local (no global IDs)
- Parallel meshes use shared entities only (no ghosting)
- Mesh entities may not be stored contiguously in memory (although they often are)

The mesh is an **adjacency based** data structure. The fundamental operations of the data structure are retrieving mesh entities that are adjacent to other mesh entities, for example retrieving the verts of an element or the elements connected to an edge. It is also a **complete** mesh representation, meaning all entitites (verts, edges, and elements) exist in the data structure.

The mesh data structure also supports attaching data to mesh entities via fields. There are presently two types of fields: `stk::middle_mesh::mesh::Field` and `stk::middle_mesh::mesh::VariableSizeField`. The regular `Field` stores a fixed number of values at each node, while `VariableSizeField` allows storing a different number of values at each node in the field. `VariableSizeField` also uses more memory than `Field` and can have worse cache locality.

To simplify the code snippets in the remainder of the document, assume a `using namespace stk::middle_mesh::mesh` statement has been entered into the program previously.

### 11.1.1. *Creating a mesh*

To create a mesh:

**Listing 11.1 Example of how to create a mesh**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
MPI_Comm comm = MPI_COMM_WORLD;
std::shared_ptr<Mesh> mesh = make_empty_mesh(comm);

```

To add vertices to the mesh:

**Listing 11.2 Example of how to add vertices to a mesh**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
MeshEntityPtr v1 = mesh->create_vertex(0, 0, 0);
MeshEntityPtr v2 = mesh->create_vertex(1, 0, 0);
MeshEntityPtr v3 = mesh->create_vertex(0, 1, 0);

```

To create edges:

**Listing 11.3 Example of how to add edges to a mesh**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
MeshEntityPtr edge1 = mesh->create_edge(v1, v2);
MeshEntityPtr edge2 = mesh->create_edge(v2, v3);
MeshEntityPtr edge3 = mesh->create_edge(v3, v1);

```

Note that edges have orientation: `edge1` goes from `v1` to `v2`.

To create a triangle:

**Listing 11.4 Example of how to create a triangle from edges**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
MeshEntityPtr tri = mesh->create_triangle(edge1, edge2,  edge3,
          EntityOrientation::Standard);

```

Notice the final argument. This argument tells the mesh that `edge1` has the same orientation as the reference triangle (ie. that the first vertex of `edge1` is the first vertex of the element). From this information, the `Mesh` can determine the orientations of the remaining edges.

An alternative way to create a triangle is:

**Listing 11.5 Example of how to create a triangle from vertices**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
MeshEntityPtr tri = mesh->create_triangle_from_verts(v1, v2, v3);

```

This function creates a triangle directly from the vertices, creating any intermediate edges if they do not already exist. Unlike `Mesh::create_triangle()`, this function does not require explicit entity orientation information. Instead, the vertices must be provided in the same order as the reference element (counterclockwise, starting from the bottom left corner).

The functions `Mesh::create_quad()` and `Mesh::create_quad_from_triangles()` are available to create quads.

### 11.1.2. *Using a mesh*

To iterate over all the entities in the mesh, use the functions:

**Listing 11.6 Functions for mesh entity iteration**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
mesh->get_vertices();
mesh->get_edges();
mesh->get_elements();
mesh->get_mesh_entities(dim);  // returns same as one of the above functions
                               // depending on dim

```

which return an iterable container of `MeshEntityPtr`. Note that this container can contain `nullptr`s, so the value yielded by the iterator must be checked:

**Listing 11.7 Example mesh iteration**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
for (MeshEntityPtr vert : mesh->get_vertices())
  if (vert)
  {
    // do stuff with vert
  }

```

The `MeshEntity` class has several functions on it that describe the mesh entity:

**Listing 11.8 MeshEntity functions**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
MeshEntityType type = edge1->get_type();   // returns enum telling if this entity is a vert,
                                           // edge, tri, or quad
int localId         = edge1->get_id();     //  returns the local ID of this entity
MeshEntityPtr vert  = edge1->get_down(i);  // returns the i'th downward adjacent entity
int numVerts        = edge1->count_down(); // returns the number of downward adjacent
                                           // entities
MeshEntityPtr triUp = edge1->get_up(i);    // returns the i'th upward adjacent entity
int numEls          = edge1->count_up();   // returns the number of upward adjacent entities

```

A few notes:

The free function `int get_type_dimension(MeshEntityType)` returns the dimension of a given entity type.

Entity local IDs are dimension specific. For example, there could be both a vertex and an edge with local ID 0. Use the dimension and local ID to uniquely identify an entity. Use the `MeshEntityCompare` comparator for sorting or associative containers.

The functions `get_down()` and `get_up()` return an entity of dimension 1 lower and 1 higher, respectively, than the entity the function was called on. For example, for an edge, `get_down()` returns a vertex and `get_up()` returns an element. For vertices, the number of downward adjacencies is zero and it is an error to call this function. Similarly for `get_up()` and elements.

The function `count_down()` is often unnecessary. Once the mesh has been constructed, the number of downard adjacent entities is known from the topology (an edge has 2 verts, a triangle has 3 edges and 3 verts, etc.).

In contrast, `count_up()` is often used because the number of upward adjacent entities is not bounded for unstructured meshes (a vertex can have an unlimited number of edges connected to it in a triangular mesh).

To get adjacencies of more than one level, use the free functions:

**Listing 11.9 Mesh adjacency functions**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
std::array<MeshEntityPtr, MAX_DOWN> downEntities;
int numDown = get_downward(tri, dim, downEntities.data());

std::vector<MeshEntityPtr> upEntities;
int numUp = get_upward(v1, dimEl, upEntities);

```

Given a mesh entity, `get_downward()` ovewrites the array `down` with all the downward adjacencies of dimension `dim`. `down` must be large enough to store the number of entities. It is recommended to create a `std::array<MeshEntityPtr, MAX_DOWN>` and pass in a pointer to its data. The `MAX_DOWN` is an upper bound on the number of downward adjacencies any entity can have. The value returned by `get_downward()` is the number of entities returned in `down`.

`get_upward()` is conceptually similar, but it returns upward adjacencies. Because the number upward adjacencies is not bounded, it takes a `std::vector` which will be resized to fit the entities. `get_upward()` also returns the number of entities.

### 11.1.3. Parallel Meshes

For parallel meshes, additional information is required when an entity exists on more than one process. This information is the `RemoteSharedEntity` struct, which contains the MPI rank and local id of the entity on a different process. The other instances of the entity must be associated with the local instance of the entity. The `MeshEntity` class has several function for this:

**Listing 11.10 MeshEntity remote shared entity functions**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
// registers that the given 'RemoteSharedEntity' represents the same mesh entity as this one
v1->add_remote_shared_entity(RemoteSharedEntity{remoteRank, remoteId});

// returns the i'th 'RemoteSharedEntity'
const RemoteSharedEntity& remote = v1->get_remote_shared_entity(i);

// returns the number of remote shared entities
int numRemotes                   = v1->count_remote_shared_entities();

```

The `RemoteSharedEntity` information must be symmetric: if vertex 7 on process 0 has a `RemoteSharedEntity` of vertex 3 on process 1, then vertex 3 on process 1 must have a

`RemoteSharedEntity` of vertex 7 on process 0. In general, every instance of a shared entity must know about every other instance of the shared entity.

The free functions:

**Listing 11.11 MeshEntity remote shared entity free functions**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
  int ownerRank = get_owner(mesh, v1);
  RemoteSharedEntity remote2 = get_remote_shared_entity(v1, remoteRank);

```

return the MPI rank of the owner of a given mesh entity and the `RemoteSharedEntity` on a given rank. The latter function throws an exception if it does not exist.

For shared edges, there is one additional requirement: the orientation of the edge must be the same on both processes (ie. the vertices that define the edge must be in the same order on both processes).

### *11.1.4. Creating and using Fields*

Fields allow storing data associated with mesh entities. They are similar to 3 dimensional arrays and can be indexed using
`operator()(MeshEntityPtr e, int node, int component).`

The `FieldShape` object defines how many nodes are each dimension entity.
`FieldShape(1, 0, 0)` has 1 node on each vertex and 0 nodes on edges and elements.
`FieldShape(0, 0, 3)` has 3 nodes on each element and zero on vertices and edges.

Fields can be created with:

**Listing 11.12 Mesh Field creation**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
FieldPtr<double> field = create_field<double>(mesh, FieldShape(1, 0, 0), componentsPerNode,
          init);

```

The `componentPerNode` arguments allows storing several values at each node, and the `init` arguments gives the initial value for the field.

To store the solution of the Navier-Stokes equations, for example, at the quadrature nodes of an element using a 3 point quadrature rule, the field would be

**Listing 11.13 Mesh Field creation with several nodes and components per node**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
  FieldPtr<double> field = create_field<double>(mesh, FieldShape(0, 0, 3), 5);

```

A `FieldPtr<T>` is a shared pointer, so it does not have to be manually freed.

Because fields are managed by `shared_ptr`, but `operator()` is used to index the field, the standard idiom is to:

**Listing 11.14 Mesh Field access idiom**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
void foo(std::shared_ptr<Mesh> mesh, FieldPtr<double> fieldPtr)
{
  auto& field = *fieldPtr;
  for (MeshEntityPtr vert : mesh->get_vertices())
    if (vert)
    {
      double val = field(vert, 0, 0);
      // do something with val
      std::cout << "field value for vert " << vert << " = " << val << std::endl;
    }
}
```

Note that fields can be created at any time during the `Mesh`'s lifetime. If new mesh entities are created, the field will automatically grow. This may result in a reallocation of the storage underlying the field, so users should avoid keeping pointers or references to field data.

### 11.1.5. Creating and using VariableSizeField

`VariableSizeField` is another type of field that allows each entity to have a different number of components per node. The tradeoff for this flexibility is increased memory usage and possibly reduced cache locality.

To create a variable sized field:

**Listing 11.15 Mesh VariableSizeField creation**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
  VariableSizeFieldPtr<double> variField = create_variable_size_field<double>(mesh,
           FieldShape(1, 0, 0));

```

Unlike a regular `Field`, there is no `componentsPerNode` argument, and no initializer, because the field is initially empty.

To insert a new value into the field, do:

**Listing 11.16 Mesh VariableSizeField value insertion**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
    variField->insert(v1, node, val);

```

This will append `val` to the other values at the given `node`, increasing the components per node by 1.

The current number of components per node can be retrieved via

**Listing 11.17 Mesh VariableSizeField value insertion**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
    int numComp = variField->get_num_comp(v1, node);

```

To retrieve or modify existing values,
`operator()(MeshEntityPtr entity, int node, int component)` can be used just like regular `Field`, however it cannot be used to append new values.

**Listing 11.18 Mesh VariableSizeField acess**
**code/stk/stk_doc_tests/stk_middle_mesh/mesh_doc_test.cpp**

```
    auto& variFieldRef = *field;
    double val2 = variFieldRef(v1, node, comp);

```

This page intentionally left blank.

# 12. STK EXPREVAL

## 12.1. Expression Evaluation

String function evaluation is handled using the STK expression evaluator. The specific set of valid variables that can be used in the function depends on where it is being used, but often includes things like time ($t$) and spatial coordinates ($x$,$y$,$z$). In some cases, global variables are also registered as valid variables.

In addition to the registered variables, there are a large number of standard functions and operations available in the evaluator. The following section shows a number of different operators with examples and descriptions; explicit usage documentation will follow in Section 12.1.2.

### *12.1.1. Types of Expressions*

#### 12.1.1.1. Powers

Powers can be specified either using the `pow` function or the standard `a^b` notation.

```
3^2
pow(3,2)
```

#### 12.1.1.2. Min and Max Functions

There are min and max functions that take 2 to 4 comma-separated inputs.

```
min(1,2)
min(1,2,3,4)
max(1,2)
max(1,2,3,4)
```

#### 12.1.1.3. Inline Variables

You can define variables inline in the function string, although doing this with aprepro in your input file is often more straightforward.

```
a=1;b=2;a+b+4
```

#### 12.1.1.4. Ramps and Pulses

There are a number of pre-defined ramp and step functions.

```
cycloidal_ramp(t,ts,te)
haversine_pulse(t,ts,te)
cosine_ramp(t)
cosine_ramp(t,te)
cosine_ramp(t,ts,te)
sign(x)
unit_step(t,ts,te)
point2d(x,y,r,w)
point3d(x,y,z,r,w)
```

**12.1.1.4.1. Cosine Ramp** The `cosine_ramp` function provides a smooth ramp from 0 to 1. It can be used with 1, 2, or 3 inputs. The form with one input uses a start and end time of 0 and 1. The form with two inputs specifies the end time, but uses a start time of 0. Note that `cos_ramp` is also a valid function, and is equivalent to `cosine_ramp`.

$$\text{cosine\_ramp}t, t_s, t_e = \begin{cases} 0, & t \leq t_s \\ \frac{1}{2}\left(1 - \cos\left(\frac{\pi t - t_s}{t_e - t_s}\right)\right), & t_s < t < t_e \\ 1, & t \geq t_e \end{cases} \tag{12.1}$$

**12.1.1.4.2. Cycloidal Ramp** The `cycloidal_ramp` function is another smooth ramp function from 0 to 1. Unlike the `cosine_ramp` function, it requires all three arguments.

$$\text{cycloidal\_ramp}t, t_s, t_e = \begin{cases} 0, & t \leq t_s \\ \frac{t - t_s}{t_e - t_s} - \frac{1}{2\pi}\sin\left(\frac{2\pi t - t_s}{t_e - t_s}\right), & t_s < t < t_e \\ 1, & t \geq t_e \end{cases} \tag{12.2}$$

**12.1.1.4.3. Haversine Pulse** The `haversine_pulse` function is a smooth sinusoidal finite width pulse, defined by

$$\text{haversine\_pulse}t, t_s, t_e = \begin{cases} 0, & t \leq t_s \\ \sin\left(\frac{\pi t - t_s}{t_e - t_s}\right)^2, & t_s < t < t_e \\ 0, & t \geq t_e \end{cases} \tag{12.3}$$

**12.1.1.4.4. Sign** The `sign` function returns the sign of its single argument

$$\mathrm{sign} x = \begin{cases} 1, & x \geq 0 \\ -1, & x < 0 \end{cases} \tag{12.4}$$

**12.1.1.4.5. Unit Step** The `unit_step` function is a sharp square step function

$$\mathrm{unit\_step} t, t_s, t_e = \begin{cases} 0, & t < t_s \\ 1, & t_s \leq t \leq t_e \\ 0, & t > t_e \end{cases} \tag{12.5}$$

**12.1.1.4.6. 2D Point** The `point2d` function provides a 2D point mask for applying a function in a certain spatial window. It is equivalent to a longer call to `cosine_ramp` where

$$\mathrm{point2d} x, y, r, w = 1 - \mathrm{cosine\_ramp} \sqrt{x^2\ y^2}, r - w2, r\ w2 \tag{12.6}$$

**12.1.1.4.7. 3D Point** The `point3d` function provides a 3D point mask for applying a function in a certain spatial window. It is equivalent to a longer call to `cosine_ramp` where

$$\mathrm{point3d} x, y, z, r, w = 1 - \mathrm{cosine\_ramp} \sqrt{x^2\ y^2\ z^2}, r - w2, r\ w2 \tag{12.7}$$

#### 12.1.1.5. Basic Math

There are standard mathematical functions. Most of these require no explanation, although it is worth noting that `log` is the natural log (equivalent to `ln` not `log10`).

```
exp(x)
ln(x)
log(x)
log10(x)
pow(a,b)
sqrt(x)
erfc(x)
erf(x)
acos(x)
asin(x)
asinh(x)
atan(x)
atan2(y,x)
atanh(x)
cos(x)
cosh(x)
```

```
acosh(x)
sin(x)
sinh(x)
tan(x)
tanh(x)
```

#### 12.1.1.6. Rounding Functions

There are a variety of rounding and numerical manipulation routines available.

```
ceil(x)
floor(x)
abs(x)
fabs(x)
mod(x,y)
ipart(x)
fpart(x)
```

You can round up with `ceil` and round down with `floor`. You can separate a number into its integer part (`ipart`) and floating point part (`fpart`). You can compute the modulus with `mod` and get the absolute value with `abs`.

#### 12.1.1.7. Polar Coordinate and Angle Helpers

When working with polar and rectangular coordinates there are helper functions for converting coordinate systems as well as for converting degrees to radians and back.

```
poltorectx(r,theta)
poltorecty(r,theta)
deg(r)
rad(d)
recttopola(x,y)
recttopolr(x,y)
```

The conversion functions are:

$$\mathrm{poltorectx} r, \theta = r \cos\theta \tag{12.8}$$

$$\mathrm{poltorecty} r, \theta = r \sin\theta \tag{12.9}$$

$$\mathrm{recttopola} x, y = \mathrm{atan2} y, x \tag{12.10}$$

$$\text{recttopolr}x, y = \sqrt{x^2\ y^2} \tag{12.11}$$

The angle returned by `recttopola` is adjusted to be between 0 and $2\pi$.

#### 12.1.1.8. Distributions and Random Sampling

There are several functions available for generating pseudo-random output with different distributions. When using random output in a string function, be careful using it in places where it can affect nonlinear convergence.

For example, setting an initial condition using a `random` distribution is acceptable since it is only evaluated once. However, using that to define a material property (for example, thermal conductivity) or boundary condition would result in that property varying not just in time and space, but also from one nonlinear iteration to the next, which would general keep a Newton solver from converging. The `ts_random` and `ts_normal` functions are designed to help with this problem.

```
weibull_pdf(x,shape,scale)
normal_pdf(x,mu,sigma)
gamma_pdf(x,shape,scale)
log_uniform_pdf(x,xmin,xmax)
exponential_pdf(x,beta)
random()
rand()
random(seed)
srand(seed)
time()
ts_random(t,x,y,z)
ts_normal(t,x,y,z,mu,sigma,minR,maxR)
```

**12.1.1.8.1. Weibull Distribution** The `weibull_pdf` function returns a Weibull distribution, where $\lambda$ is the scale parameter and $k$ is the shape parameter.

$$\text{weibull\_pdf}x, k, \lambda = \begin{cases} \frac{k}{\lambda}\left(\frac{x}{\lambda}\right)^{k-1} e^{-x\lambda^k}, & x \geq 0 \\ 0, & x < 0 \end{cases} \tag{12.12}$$

**12.1.1.8.2. Normal Distribution** The `normal_pdf` function returns a normal distribution, where $\mu$ is the mean and $\sigma$ is the standard deviation.

$$\text{normal\_pdf}x, \mu, \sigma = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{x-\mu^2}{2\sigma^2}\right) \tag{12.13}$$

**12.1.1.8.3. Gamma Distribution** The `gamma_pdf` function returns a gamma distribution, where $\theta$ is the scale parameter and $k$ is the shape parameter.

$$\text{gamma\_pdf}x, k, \theta = \frac{1}{\Gamma k \theta^k} x^{k-1} e^{-x\theta} \tag{12.14}$$

**12.1.1.8.4. Log Uniform Distribution** The `log_uniform_pdf` function returns a log-uniform distribution

$$\text{log\_uniform\_pdf}x, x_0, x_1 = \frac{1}{\ln x_1 - \ln x_0 x} \tag{12.15}$$

**12.1.1.8.5. Exponential Distribution** The `exponential_pdf` function returns a one-parameter exponential distribution

$$\text{exponential\_pdf}x, \beta = \frac{1}{\beta} e^{-x\beta} \tag{12.16}$$

**12.1.1.8.6. Random Values** There are several functions for generating pseudo-random numbers. These can be used as inputs to any of the distributions, or as standalone values to get a uniform distribution.

`random()` calls a platform-indepent fast pseudo-random number algorithm. It is seeded with a static constant, so the first call to it will always produce the same value. The value returned will be between 0 and 1.

`random(seed)` calls the same algorithm as `random()` but sets the seed first then returns a number between 0 and 1. The seed in this case can be any floating point number.

`time()` returns the current time integer. This can used to provide a random number seed to make the distributions non-deterministic from run to run.

`ts_random(t,x,y,z)` returns a uniformly distributed random number from 0 to 1 that is unique in time and space. This means that repeated evaluations of this function at the same point in time and space will produce the same output. This is useful for applying randomness on boundary conditions, for example, where the value should not change during nonlinear iterations at a given time.

`ts_normal(t,x,y,z,mu,sigma,minR,maxR)` returns a clipped normally distributed random number with a mean of `mu` and standard deviation of `sigma`. The output is clipped to be between `minR` and `maxR` and, like `ts_random`, the output is deterministic in time and space.

#### 12.1.1.9. Boolean and Ternary Logic

In addition to all these functions, you can make more complicated functions using ternary statements and boolean logic.

You can use ternary operators for simple conditional assignment. The basic structure of a ternary operator is `boolean ? value_if_true : value_if_false` For example, the following if-else code

```
if x > 2.2:
  return 1.2*y
else
  return 0.1*y
```

can be converted to a ternary operator that returns the same thing

```
(x > 2.2 ? 1.2 : 0.1) * y
```

You can also use boolean operations in the string functions. By default, booleans that evaluate to true are treated as 1 and booleans that are false are 0. The prior example can be expressed using boolean statements as

```
(x>2.2)*1.2*y + (x<=2.2)*0.1*y
((x>2.2)*1.1 + 0.1)*y
```

These operations can be combined (multiplication of boolean statements is equivalent to a logical `and`). To replicate the `unit_step` function described earlier you can use either ternary or boolean operators. The following three functions all produce the same result—a step of height 5 between 1 and 2.

```
5*unit_step(x,1,2)
((x>1 && x<2) ? 5 : 0)
5*(x>1)*(x<2)
```

Note that chained inequality syntax is not allowed:

```
1 < x < 2       // parse error
(1<x) && (x<2)  // correct
```

#### 12.1.1.10. User-defined functions

Users can specify their own expression functions via `Sierra::UserInputFunction`.

### *12.1.2. Usage Examples*

The STK expression evaluator can be used on both host or device. In both instances, evaluation begins with the creation of a `stk::expreval::Eval` object; this object is constructed by providing the string expression to be evaluated. It owns the two main components used in performing the expression evaluation:

1. The `VariableMap`, which is a `std::map` that stores all `Variables` that appear in the expression. Each `Variable` contains the names and values of the variables, as well as other information such as sizing and type. Users can set and modify the values of `Variables` that are in the `VariableMap`.
2. The `Node` tree, which consists of individual `Nodes` that contain the operational information (such as addition, multiplication, etc.) of the expression. The expression is evaluated through traversal of this tree. Users cannot modify `Nodes`.

After initial construction, the expression must be parsed to population the information that is contained in the `VariableMap` and `NodeTree` and prepare for evaluation. The parsing stage checks that the expression itself is correct (e.g., the expression `"x = (y+2"` would fail to parse due to unbalanced parentheses), and that all subexpressions that appear in the expression are syntactically correct (e.g., `"sing(x)"` would parse correctly, but it is syntactically incorrect because `"sing"` is not a recognized function in the expression evaluator). Examples of both successfully and unsuccessfully parsed expressions are shown in Listing 12.1.

**Listing 12.1 Example of Parsing Expressions**
**code/stk/stk_doc_tests/stk_expreval/ParsedExpressionSyntax.cpp**

```cpp
bool
isValidParse(const char *expr)
{
  stk::expreval::Eval expr_eval(expr);
  EXPECT_NO_THROW(expr_eval.parse());
  return expr_eval.getSyntaxStatus();
}

bool
isInvalidParse(const char *expr)
{
  stk::expreval::Eval expr_eval(expr);
  try {
    expr_eval.parse();
  }
  catch (std::runtime_error& ) {
    return !expr_eval.getSyntaxStatus();
  }

  return false;
}

TEST(ParsedEval, testAlgebraicSyntax)
{
  EXPECT_TRUE(isValidParse(""));
  EXPECT_TRUE(isValidParse(";;"));
  EXPECT_TRUE(isValidParse("2*2"));
  EXPECT_TRUE(isValidParse("3^2"));
  EXPECT_TRUE(isValidParse("x*-0.1"));
  EXPECT_TRUE(isValidParse("x*+0.1"));
  EXPECT_TRUE(isValidParse("x--7.0"));
  EXPECT_TRUE(isValidParse("x*-x"));
  EXPECT_TRUE(isValidParse("x*+x"));
  EXPECT_TRUE(isValidParse("v[0]=v[1]*0.1"));
  EXPECT_TRUE(isValidParse("x---x"));

  EXPECT_TRUE(isInvalidParse("0.01.02"));
  EXPECT_TRUE(isInvalidParse("5*.e+10"));
  EXPECT_TRUE(isInvalidParse("x y"));
  EXPECT_TRUE(isInvalidParse("x(y"));
  EXPECT_TRUE(isInvalidParse("x*"));
```

```
  EXPECT_TRUE(isInvalidParse("x*(y+1"));
  EXPECT_TRUE(isInvalidParse("cos(x"));
  EXPECT_TRUE(isInvalidParse("(x)y"));
  EXPECT_TRUE(isInvalidParse("()"));
}


bool
isValidFunction(const char *expr)
{
  stk::expreval::Eval expr_eval(expr);
  EXPECT_NO_THROW(expr_eval.parse());
  return !expr_eval.undefinedFunction();
}

bool
isInvalidFunction(const char *expr)
{
  stk::expreval::Eval expr_eval(expr);
  try {
    expr_eval.parse();
  }
  catch (std::runtime_error& ) {
    return expr_eval.undefinedFunction();
  }

  return false;
}

TEST(ParsedEval, testFunctionSyntax)
{
  EXPECT_TRUE(isValidFunction("sin(1)"));
  EXPECT_TRUE(isValidFunction("SIN(1)"));
  EXPECT_TRUE(isValidFunction("rand()"));
  EXPECT_TRUE(isValidFunction("time()"));
  EXPECT_TRUE(isValidFunction("random()"));
  EXPECT_TRUE(isValidFunction("random(1)"));
  EXPECT_TRUE(isValidFunction("cosine_ramp(x,y)"));
  EXPECT_TRUE(isValidFunction("normal_pdf(x, alpha, beta)"));

  EXPECT_TRUE(isInvalidFunction("stress(1)"));
  EXPECT_TRUE(isInvalidFunction("gamma(1)"));
}
```

Once the parsing stage has been successfully completed, users can query various properties of the expression:

- if an expression is constant
- whether a variable appears in the expression or not
- if a variable is a scalar
- the number of variables in the expression

Users can also retrieve the populated `VariableMap`, get a list of all variable names that appear in the expression, get a list of all dependent variable names that appear in the expression, or get a list of all dependent variable names that appear in the expression. Examples of accessing this information are shown in Listing 12.2.

**Listing 12.2 Example of Querying a Parsed Expression**
**code/stk/stk_doc_tests/stk_expreval/ParsedExpressionQueries.cpp**

```
TEST(ParsedEval, isConstantExpression)
{
  stk::expreval::Eval evalEmpty;
  evalEmpty.parse();
  EXPECT_TRUE(evalEmpty.is_constant_expression());

  stk::expreval::Eval evalConstant("2");
  evalConstant.parse();
  EXPECT_TRUE(evalConstant.is_constant_expression());

  stk::expreval::Eval evalVar("x");
  evalVar.parse();
  EXPECT_FALSE(evalVar.is_constant_expression());
}

TEST(ParsedEval, isVariable)
{
  stk::expreval::Eval evalEmpty;
  evalEmpty.parse();
  EXPECT_FALSE(evalEmpty.is_variable("x"));

  stk::expreval::Eval evalTwoVar("x + y");
  evalTwoVar.parse();
  EXPECT_TRUE(evalTwoVar.is_variable("x"));
  EXPECT_TRUE(evalTwoVar.is_variable("y"));
  EXPECT_FALSE(evalTwoVar.is_variable("z"));

  stk::expreval::Eval evalInsVar("lambda + Lambda");
  evalInsVar.parse();
  EXPECT_EQ(evalInsVar.get_variable_names().size(), 1u);
  EXPECT_TRUE(evalInsVar.is_variable("LAMBDA"));
  EXPECT_TRUE(evalInsVar.is_variable("lambda"));
  EXPECT_TRUE(evalInsVar.is_variable("Lambda"));

}

TEST(ParsedEval, isScalar)
{
  stk::expreval::Eval eval("x");
  eval.parse();
  EXPECT_TRUE(eval.is_scalar("x"));

  stk::expreval::Eval evalBind("y^2");
  evalBind.parse();
  EXPECT_TRUE(evalBind.is_scalar("y"));

  stk::expreval::Eval evalBindArray("z");
  evalBindArray.parse();
  double z[3] = {4.0, 5.0, 6.0};
  evalBindArray.bindVariable("z", *z, 3);
  EXPECT_FALSE(evalBindArray.is_scalar("z"));
}

TEST(ParsedEval, getAllVariables)
{
  stk::expreval::Eval eval;
  eval.parse();
  EXPECT_EQ(eval.get_variable_names().size(), 0u);

  stk::expreval::Eval evalVars("x = sin(y)");
  evalVars.parse();
  EXPECT_EQ(evalVars.get_variable_names().size(), 2u);
  EXPECT_TRUE(evalVars.is_variable("x"));
  EXPECT_TRUE(evalVars.is_variable("y"));
}

TEST(ParsedEval, getDependentVariables)
{
```

```
  stk::expreval::Eval eval("x");
  eval.parse();
  EXPECT_EQ(eval.get_dependent_variable_names().size(), 0u);

  stk::expreval::Eval evalAssign("x = 2");
  evalAssign.parse();
  EXPECT_EQ(evalAssign.get_dependent_variable_names().size(), 1u);
  EXPECT_TRUE(evalAssign.is_variable("x"));

  stk::expreval::Eval evalTwoVar("x = 2; y = x");
  evalTwoVar.parse();
  EXPECT_EQ(evalTwoVar.get_dependent_variable_names().size(), 2u);
  EXPECT_TRUE(evalTwoVar.is_variable("x"));
  EXPECT_TRUE(evalTwoVar.is_variable("y"));
}

TEST(ParsedEval, getIndependentVariables)
{
  stk::expreval::Eval eval("x");
  eval.parse();
  EXPECT_EQ(eval.get_independent_variable_names().size(), 1u);
  EXPECT_TRUE(eval.is_variable("x"));

  stk::expreval::Eval evalAssign("x = 2");
  evalAssign.parse();
  EXPECT_EQ(evalAssign.get_independent_variable_names().size(), 0u);
  EXPECT_TRUE(evalAssign.is_variable("x"));

  stk::expreval::Eval evalTwoVar("x = sin(y)");
  evalTwoVar.parse();
  EXPECT_EQ(evalTwoVar.get_independent_variable_names().size(), 1u);
  EXPECT_TRUE(evalTwoVar.is_variable("x"));
  EXPECT_TRUE(evalTwoVar.is_variable("y"));
}
```

When `Variables` are identified in the parsed expression, they are assumed to be scalar and are assigned a default value of zero. Once the expression has been parsed (and before the expression has been evaluated), users can override this default value and assign, or "bind", values to `Variables` from their own data (such as time-step data, model coefficients, etc.). Though variables are assumed to be scalar, it is possible to bind arrays to variables, as long as the array sizing and indexing are consistent with the expression. The option to use zero-based or one-based indexing for array variables is set during construction of the `stk::expreval::Eval` object; zero-based indexing is the default.

It is also possible to unbind the `Variable`'s value, resetting it to the original default `Variable`, as well as deactivate it so that this variable can no longer be used in the evaluation expression (this results in a throw). This can be used to help prevent out-of-date data from being used in expression evaluation. Listing 12.3 demonstrates some of these `Variable` properties.

**Listing 12.3 Examples of Different Types and States of Variables**
**code/stk/stk_doc_tests/stk_expreval/VariableStates.cpp**

```
TEST(Variable, scalar_vs_array)
{
  stk::expreval::Eval expr("x[1]", stk::expreval::Variable::ArrayOffset::ZERO_BASED_INDEX);
  expr.parse();
  EXPECT_TRUE(expr.is_scalar("x"));
  EXPECT_EQ(expr.getValue("x"), 0.0);
  EXPECT_ANY_THROW(expr.evaluate());

  double x[2] = {3.0, 4.0};
```

```
  expr.bindVariable("x", *x, 2);
  EXPECT_FALSE(expr.is_scalar("x"));
  EXPECT_EQ(expr.evaluate(), 4.0);

  stk::expreval::Eval expr2("y[1]", stk::expreval::Variable::ArrayOffset::ONE_BASED_INDEX);
  expr2.parse();
  double y[2] = {3.0, 4.0};
  expr2.bindVariable("y", *y, 2);
  EXPECT_FALSE(expr2.is_scalar("y"));
  EXPECT_EQ(expr2.evaluate(), 3.0);
}

TEST(Variable, demonstrate_states)
{
  stk::expreval::Eval expr("x");
  expr.parse();
  EXPECT_EQ(expr.evaluate(), 0.0);

  double x = 2.0;
  expr.bindVariable("x", x, 1);
  EXPECT_EQ(expr.evaluate(), 2.0);

  expr.unbindVariable("x");
  EXPECT_EQ(expr.evaluate(), 0.0);

  expr.deactivateVariable("x");
  EXPECT_ANY_THROW(expr.evaluate());
}
```

Once all `Variable` data has been assigned, the expression can be evaluated. This results in a `double` value that is returned to the user. Examples for this stage of the expression evaluation will be shown in the following sections on host-side and device-side expression evaluation, since the procedure differs slightly for the two.

#### 12.1.2.1. Host Expression Evaluation

Expression evaluation on the host is straightforward and consists of four sequentially-executed steps: creation, parsing, variable value assignment, and final evaluation. These steps are denoted in Figure 12-1, with basic examples shown in Listing 12.4 and more complex examples shown in Listing 12.5.

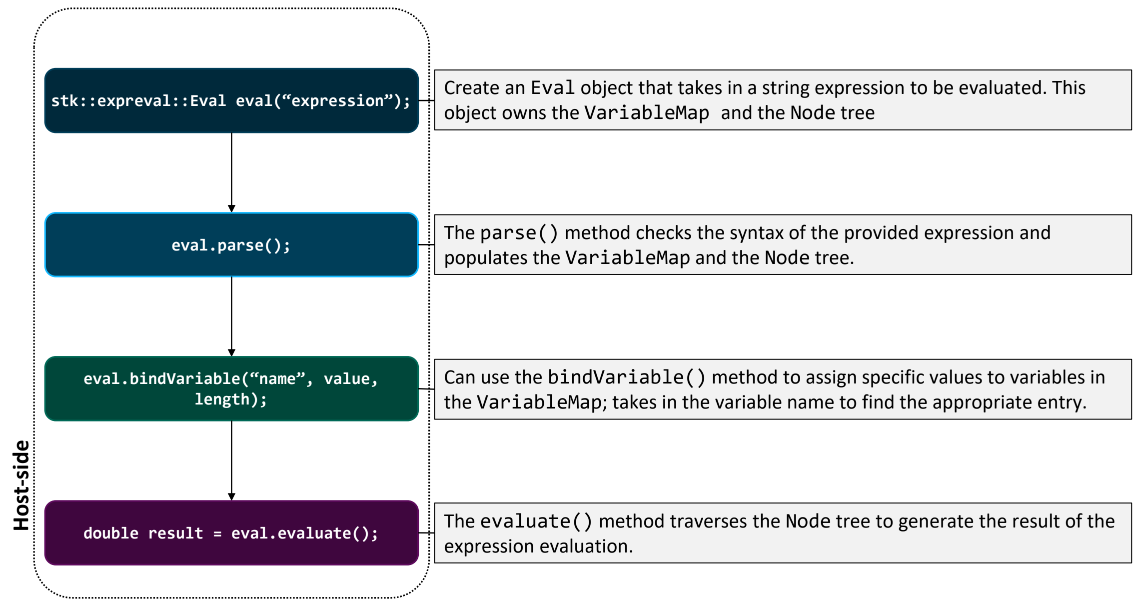



**Figure 12-1. Host-side Expression Evaluation**

**Listing 12.4 Evaluation of Basic Operations and Functions**
**code/stk/stk_doc_tests/stk_expreval/BasicHostEvaluation.cpp**

```cpp
double evaluate(const std::string & expression)
{
  stk::expreval::Eval eval(expression);
  eval.parse();
  return eval.evaluate();
}

TEST(HostEvaluation, testOpcodes)
{
  EXPECT_DOUBLE_EQ(evaluate(""), 0.0);
  EXPECT_DOUBLE_EQ(evaluate("-1.333"), -1.333);
  EXPECT_DOUBLE_EQ(evaluate("(1+4+9+16)+(9+16+25+36)"), 116);
  EXPECT_DOUBLE_EQ(evaluate("(-1-4)-(9-16)"), 2);
  EXPECT_DOUBLE_EQ(evaluate("2*3*4*5"),      120);
  EXPECT_DOUBLE_EQ(evaluate("(120/5)/(4/3)"), 18);
  EXPECT_DOUBLE_EQ(evaluate("---2"),         -2);
  EXPECT_DOUBLE_EQ(evaluate("9.1 % 4"),     1.1);
  EXPECT_DOUBLE_EQ(evaluate("3^2^2"),        81);
  EXPECT_DOUBLE_EQ(evaluate("0.1==0.999999"), 0);
  EXPECT_DOUBLE_EQ(evaluate("2!=(1+1)"),      0);
  EXPECT_DOUBLE_EQ(evaluate("1<1.000001"), 1);
  EXPECT_DOUBLE_EQ(evaluate("1>1"),        0);
  EXPECT_DOUBLE_EQ(evaluate("1<=1"),       1);
  EXPECT_DOUBLE_EQ(evaluate("2>=(1+2)"),   0);
  EXPECT_ANY_THROW(evaluate("1 <= 2 < 3"));
  EXPECT_DOUBLE_EQ(evaluate("!0"),        1);
  EXPECT_DOUBLE_EQ(evaluate("!0.000001"), 0);
  EXPECT_DOUBLE_EQ(evaluate("0 && 0"),        0);
  EXPECT_DOUBLE_EQ(evaluate("0 && 1"),        0);
  EXPECT_DOUBLE_EQ(evaluate("0 || 0"),        0);
  EXPECT_DOUBLE_EQ(evaluate("0 || 1"),        1);
  EXPECT_DOUBLE_EQ(evaluate("0 ? 1 : (1+1)"),   2);
```

```
  EXPECT_DOUBLE_EQ(evaluate("x=1; y=2; z=3"),   3);
  EXPECT_DOUBLE_EQ(evaluate("x=1; y=2; x+y"),   3);
  EXPECT_DOUBLE_EQ(evaluate("(1+2+3+4)^(1+1)"), 100);
  EXPECT_DOUBLE_EQ(evaluate("15%(1+1+1)"),      0);
  EXPECT_DOUBLE_EQ(evaluate("x + y + z"),        0);
  EXPECT_DOUBLE_EQ(evaluate("x[0]"),             0);
  EXPECT_ANY_THROW(evaluate("x[0]+x[1]+x[2]"));
}

TEST(HostEvaluation, testFunctions)
{
  EXPECT_DOUBLE_EQ(evaluate("abs(-2*3)"), 6);
  EXPECT_DOUBLE_EQ(evaluate("fabs(1.5)"),  1.5);
  EXPECT_DOUBLE_EQ(evaluate("max(-1,-2,-3)"),    -1);
  EXPECT_DOUBLE_EQ(evaluate("min(3+2,2+1)"),      3);
  EXPECT_DOUBLE_EQ(evaluate("sign(-0.5)"),      -1);
  EXPECT_DOUBLE_EQ(evaluate("ipart(2.5)"),        2);
  EXPECT_DOUBLE_EQ(evaluate("fpart(-2.5)"),      -0.5);
  EXPECT_DOUBLE_EQ(evaluate("ceil(-0.999999)"),  0);
  EXPECT_DOUBLE_EQ(evaluate("floor(-0.000001)"), -1);
  EXPECT_DOUBLE_EQ(evaluate("mod(9, -4)"),   1);
  EXPECT_DOUBLE_EQ(evaluate("fmod(9, 3.5)"),  2);
  EXPECT_DOUBLE_EQ(evaluate("pow(3, 2.5)"),               std::pow(3, 2.5));
  EXPECT_DOUBLE_EQ(evaluate("sqrt(1.21)"), 1.1);
  EXPECT_DOUBLE_EQ(evaluate("exp(1.5)"), std::exp(1.5));
  EXPECT_DOUBLE_EQ(evaluate("ln(exp(1.5))"), 1.5);
  EXPECT_DOUBLE_EQ(evaluate("log(0.5)"),      std::log(0.5));
  EXPECT_DOUBLE_EQ(evaluate("log10(1)"),        0);
  EXPECT_DOUBLE_EQ(evaluate("deg(PI/4)"),     45);
  EXPECT_DOUBLE_EQ(evaluate("rad(45)"),    stk::expreval::pi()/4);
  EXPECT_DOUBLE_EQ(evaluate("sin(PI/4)"),    std::sqrt(2)/2);
  EXPECT_DOUBLE_EQ(evaluate("cos(PI/4)"),    std::sqrt(2)/2);
  EXPECT_DOUBLE_EQ(evaluate("tan(PI/4)"),    1);
  EXPECT_DOUBLE_EQ(evaluate("asin(sqrt(2)/2)"), stk::expreval::pi()/4);
  EXPECT_DOUBLE_EQ(evaluate("acos(sqrt(2)/2)"), stk::expreval::pi()/4);
  EXPECT_DOUBLE_EQ(evaluate("atan(1)"),       stk::expreval::pi()/4);
  EXPECT_DOUBLE_EQ(evaluate("atan2(1, 1)"),    stk::expreval::pi()/4);
  EXPECT_DOUBLE_EQ(evaluate("sinh(0.5)"),  std::sinh(0.5));
  EXPECT_DOUBLE_EQ(evaluate("cosh(0.5)"),  std::cosh(0.5));
  EXPECT_DOUBLE_EQ(evaluate("tanh(0.5)"),  std::tanh(0.5));
  EXPECT_DOUBLE_EQ(evaluate("asinh(0.5)"),  std::asinh(0.5));
  EXPECT_DOUBLE_EQ(evaluate("acosh(2)"),    std::acosh(2));
  EXPECT_DOUBLE_EQ(evaluate("atanh(0.5)"),  std::atanh(0.5));
  EXPECT_DOUBLE_EQ(evaluate("erf(-1)"),   std::erf(-1));
  EXPECT_DOUBLE_EQ(evaluate("erfc(-1.5)"), std::erfc(-1.5));
  EXPECT_DOUBLE_EQ(evaluate("poltorectx(5, PI)"),    -5);
  EXPECT_DOUBLE_EQ(evaluate("poltorecty(5, PI)"),      5*std::sin(stk::expreval::pi()));
  EXPECT_DOUBLE_EQ(evaluate("recttopolr(-1, 0)"), 1);
  EXPECT_DOUBLE_EQ(evaluate("recttopola(-1, 0)"),  stk::expreval::pi());
  EXPECT_DOUBLE_EQ(evaluate("unit_step(0.5, 0, 1)"),   1);
  EXPECT_DOUBLE_EQ(evaluate("cycloidal_ramp(1, 0, 1)"),     1);
  EXPECT_DOUBLE_EQ(evaluate("cos_ramp(1/3, 0, 1)"),  0.25);
  EXPECT_DOUBLE_EQ(evaluate("cos_ramp(1/3, 1)"),  0.25);
  EXPECT_DOUBLE_EQ(evaluate("cos_ramp(1/3)"),  0.25);
  EXPECT_DOUBLE_EQ(evaluate("cosine_ramp(1/3, 0, 1)"),  0.25);
  EXPECT_DOUBLE_EQ(evaluate("cosine_ramp(1/3, 1)"),  0.25);
  EXPECT_DOUBLE_EQ(evaluate("cosine_ramp(1/3)"),  0.25);
  EXPECT_DOUBLE_EQ(evaluate("haversine_pulse(1/6, 0, 1)"),  0.25);
  EXPECT_DOUBLE_EQ(evaluate("point2d(1, 0, 1, 1)"),   0.5);
  EXPECT_DOUBLE_EQ(evaluate("point3d(0, -1, 0, 1, 1)"),   0.5);
}

double reference_normal_pdf(double x, double mu, double sigma) {
  return std::exp(-(x-mu)*(x-mu)/(2.0*sigma*sigma)) /
            std::sqrt(2.0*stk::expreval::pi()*sigma*sigma);
}

double reference_weibull_pdf(double x, double k, double lambda) {
```

```
  return (x >= 0) ? (k/lambda)*std::pow(x/lambda, k-1)*std::exp(-std::pow(x/lambda, k)) : 0;
}

double reference_gamma_pdf(double x, double k, double theta) {
  return (x >= 0) ? 1/(std::tgamma(k)*std::pow(theta, k))*std::pow(x, k-1)*std::exp(-x/theta)
            : 0;
}

TEST(HostEvaluation, testPDFFunctions)
{
  EXPECT_DOUBLE_EQ(evaluate("exponential_pdf(0, 1)"),  1);
  EXPECT_DOUBLE_EQ(evaluate("log_uniform_pdf(2, 1, E)"),   0.5);
  EXPECT_DOUBLE_EQ(evaluate("normal_pdf(0.75, 1, 0.5)"), reference_normal_pdf(0.75, 1, 0.5));
  EXPECT_DOUBLE_EQ(evaluate("weibull_pdf(1, 5, 1)"),    reference_weibull_pdf(1, 5, 1));
  EXPECT_DOUBLE_EQ(evaluate("gamma_pdf(5, 5, 1)"),   reference_gamma_pdf(5, 5, 1));
}
```

**Listing 12.5 Evaluation of Bound Variables on the Host**
**code/stk/stk_doc_tests/stk_expreval/BoundHostEvaluation.cpp**

```
TEST(HostEvaluation, bindScalar)
{
  stk::expreval::Eval expr("x=5; y=y+x; y+z");
  expr.parse();
  double y = 3.0;
  double z = 4.0;
  expr.bindVariable("y", y, 1);
  expr.bindVariable("z", z, 1);
  EXPECT_DOUBLE_EQ(expr.evaluate(), 12);
}

TEST(HostEvaluation, bindVector)
{
  stk::expreval::Eval expr("(a[0]*b[0] + a[1]*b[1] + a[2]*b[2])^0.5");
  expr.parse();
  double a[3] = {1, 2, 3};
  double b[3] = {5, 4, 4};
  expr.bindVariable("a", *a, 3);
  expr.bindVariable("b", *b, 3);
  EXPECT_DOUBLE_EQ(expr.evaluate(), 5);
}

TEST(HostEvaluation, bindVectorOneBasedIndex)
{
  stk::expreval::Eval expr("(a[1]*b[1] + a[2]*b[2] + a[3]*b[3])^0.5",
            stk::expreval::Variable::ONE_BASED_INDEX);
  expr.parse();
  double a[3] = {1, 2, 3};
  double b[3] = {5, 4, 4};
  expr.bindVariable("a", *a, 3);
  expr.bindVariable("b", *b, 3);
  EXPECT_DOUBLE_EQ(expr.evaluate(), 5);
}
```

#### 12.1.2.2. Device Expression Evaluation

Device-side expression evaluation is more involved than host-side evaluation because there are data type limitations on the GPU (for example, string expressions and `std::map`, which are crucial to the initial setup stage, cannot be used). Therefore, device-side expression evaluation consists of two stages: a host-side stage, which creates the stk::expreval::Eval object, parses it, and

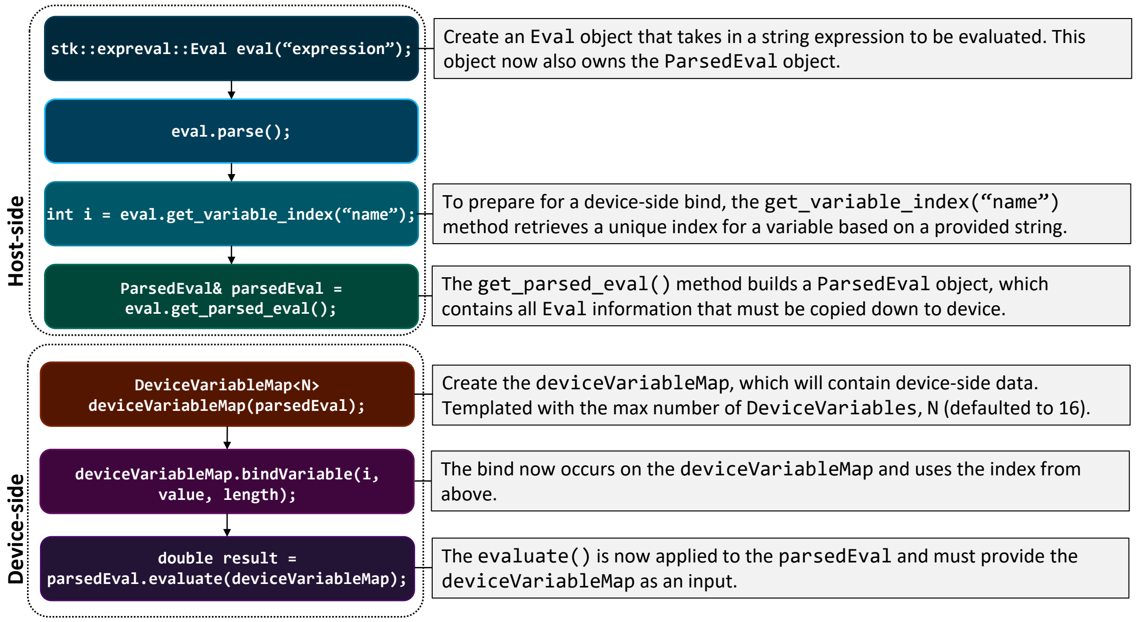



**Figure 12-2. Device-side Expression Evaluation**

**prepares the necessary data that needs to be sent to device**, and the device-side stage, which uses this stripped-down data to then perform the actual expression evaluation. These steps are shown in Figure 12-2, and usage examples are shown in Listing 12.6.

Memory on the device is often at a premium, especially for large, long-running mulitphysics applications. Because of this, it is important that the expression evaluator use as little memory as possible when completing its task on device. Since users employ the expression evaluator for a wide complexity of expressions, there are two main sizing options that default to smaller values but can be increased if needed. The `ParsedEval` object contains a sizing option for the number of entries in the buffer used to store temporary results when evaluating the expression; this is set via the `RESULT_BUFFER_SIZE` template option, and defaults to 16 entries. There is also a sizing option for the `DeviceVariableMap`, which is the main data structure that drives expression evaluation on the device. It is constructed with a template argument, `MAX_BOUND_VARIABLES`, which accounts for the number of variables in the expression that will be bound with device-side data. It also defaults to 16 entries. If these default values are not sufficient for a provided expression, a throw message with details about the sizing discrepancy is provided.

**Listing 12.6 Evaluation of Bound Variables on the Device**
**code/stk/stk_doc_tests/stk_expreval/BoundDeviceEvaluation.cpp**

```

using ViewInt1DHostType = Kokkos::View<int*, Kokkos::LayoutRight, Kokkos::HostSpace>;

double perform_device_evaluation(const std::string& expression)
{
  stk::expreval::Eval eval(expression);
  eval.parse();

```

```
  //For device-side variable binding and evaluation, need to generate a unique index for each
            variable.
  int yIndex = eval.get_variable_index("y");
  int zIndex = eval.get_variable_index("z");

  //create ParsedEval that holds all necessary info for device
  auto & parsedEval = eval.get_parsed_eval();

  //evaluate the expression on device
  double result = 0.0;
  Kokkos::parallel_reduce(stk::ngp::DeviceRangePolicy(0, 1),
    KOKKOS_LAMBDA (const int& i, double& localResult) {

      //device data that will be bound to expression variables
      double yDeviceValue = 3.0;
      double zDeviceValue = 4.0;

      //create DeviceVariableMap, which will contain device-side data
      stk::expreval::DeviceVariableMap<> deviceVariableMap(parsedEval);

      //bind variable values via the DeviceVariableMap
      deviceVariableMap.bind(yIndex, yDeviceValue, 1, 1);
      deviceVariableMap.bind(zIndex, zDeviceValue, 1, 1);

      localResult = parsedEval.evaluate(deviceVariableMap);

    }, result);

  return result;
}

TEST(DeviceEvaluation, bindScalar)
{
  double result = perform_device_evaluation("x=5; y=y+x; y+z");
  EXPECT_DOUBLE_EQ(result, 12);
}
```

**12.1.2.2.1. Limitations** The following functions cannot be used in device-side evaluation:

- `rand()`
- `srand(seed)`
- `time()`
- `random()`, `random(seed)`
- user-defined functions (`Sierra::UserInputFunctions`)

This page intentionally left blank.

## BIBLIOGRAPHY


[1] Larry A. Schoof and Victor R. Yarberry, "EXODUSII: A Finite Element Data Model," SAND92-2137, Sandia National Laboratories, Albuquerque, NM, September, 1994.[1]

[2] C. Farhat, K. G. Van der Zee, and P. Geuzaine "Provably second-order time-accurate loosely-coupled solution algorithms for transient nonlinear computational aeroelasticity," *Computer Methods in Applied Mechanics and Engineering*, 2006; 195 (17-18): 1973-2001.

[3] H. C. Edwards, A. B. Williams, G. Sjaardema, D. Baur, W. Cochran, "Sierra Toolkit Computational Mesh Conceptual Model," SAND2010-1192, Sandia National Laboratories, Albuquerque, NM, March, 2010.

[4] Jon L. Bentley "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, 1975; 18 (9): 509-517.

[5] Mark de Berg, Mark van Krewald, Mark Overmars, and Otfried Schwarzkopf *Computational Geometry: Algorithms and Applications (2nd, revised edition)*, Springer-Verlag, 2000.


[1] This document is very out of date. A new document is being prepared and a draft of the current state is available at `http://jal.sandia.gov/SEACAS/Documentation/exodusII-new.pdf`.

This page intentionally left blank.

# INDEX

This page intentionally left blank.

## DISTRIBUTION

### Email—Internal

| Name | Org. | Sandia Email Address |
|---|---|---|
| Technical Library | 1911 | sanddocs@sandia.gov |

This page intentionally 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.