```
#include "gridpack/include/gridpack.hpp"

#include "gridpack/applications/modules/powerflow/pf_app_module.hpp"

#include "gridpack/applications/contingency_analysis/ca_driver.hpp"


// Sets up multiple communicators so that individual contingency calculations
// can be run concurrently



Basic constructor


gridpack::contingency_analysis::CADriver::CADriver(void)
{
}



Basic destructor


gridpack::contingency_analysis::CADriver::~CADriver(void)
{
}



Get list of contingencies from external file
@param cursor pointer to contingencies in input deck
@return vector of contingencies


std::vector<gridpack::powerflow::Contingency>
  gridpack::contingency_analysis::CADriver::getContingencies(
      gridpack::utility::Configuration::ChildCursors contingencies)
{
  // The contingencies ChildCursors argument is a vector of configuration
// pointers. Each element in the vector is pointing at a seperate Contingency
// block within the Contingencies block in the input file.
std::vector<gridpack::powerflow::Contingency> ret;
int size = contingencies.size();
int i, idx;
// Create string utilities object to help parse file
gridpack::utility::StringUtils utils;
// Loop over all child cursors
for (idx = 0; idx < size; idx++) {
  std::string ca_type;
  contingencies[idx]->get("contingencyType",&ca_type);
  // Contingency name is used to direct output to different files for each
  // contingency
  std::string ca_name;
  contingencies[idx]->get("contingencyName",&ca_name);
  if (ca_type == "Line") {
    std::string buses;
    contingencies[idx]->get("contingencyLineBuses",&buses);
    std::string names;
    contingencies[idx]->get("contingencyLineNames",&names);
    // Tokenize bus string to get a list of individual buses
    std::vector<std::string> string_vec = utils.blankTokenizer(buses);
    // Convert buses from character strings to ints
    std::vector<int> bus_ids;
    for (i=0; i<string_vec.size(); i++) {
      bus_ids.push_back(atoi(string_vec[i].c_str()));
    }
    string_vec.clear();
    // Tokenize names string to get a list of individual line tags
    string_vec = utils.blankTokenizer(names);
    std::vector<std::string> line_names;
    // clean up line tags so that they are exactly two characters
    for (i=0; i<string_vec.size(); i++) {
      line_names.push_back(utils.clean2Char(string_vec[i]));
    }
    // Check to make sure we found everything
    if (bus_ids.size() == 2*line_names.size()) {
      // Add contingency parameters to contingency struct
      gridpack::powerflow::Contingency contingency;
      contingency.p_name = ca_name;
      contingency.p_type = Branch;
      int i;
      for (i = 0; i < line_names.size(); i++) {
        contingency.p_from.push_back(bus_ids[2*i]);
        contingency.p_to.push_back(bus_ids[2*i+1]);
        contingency.p_ckt.push_back(line_names[i]);
        contingency.p_saveLineStatus.push_back(true);
      }
      // Add branch contingency to contingency list
      ret.push_back(contingency);
    }
  } else if (ca_type == "Generator") {
    std::string buses;
    contingencies[idx]->get("contingencyBuses",&buses);
    std::string gens;
    contingencies[idx]->get("contingencyGenerators",&gens);
    // Tokenize bus string to get a list of individual buses
    std::vector<std::string> string_vec = utils.blankTokenizer(buses);
    std::vector<int> bus_ids;
    // Convert buses from character strings to ints
    for (i=0; i<string_vec.size(); i++) {
      bus_ids.push_back(atoi(string_vec[i].c_str()));
    }
    string_vec.clear();
    // Tokenize gens string to get a list of individual generator tags
    string_vec = utils.blankTokenizer(gens);
    std::vector<std::string> gen_ids;
    // clean up generator tags so that they are exactly two characters
    for (i=0; i<string_vec.size(); i++) {
      gen_ids.push_back(utils.clean2Char(string_vec[i]));
    }
    // Check to make sure we found everything
    if (bus_ids.size() == gen_ids.size()) {
      gridpack::powerflow::Contingency contingency;
      contingency.p_name = ca_name;
      contingency.p_type = Generator;
```

```
        int i;
        for (i = 0; i < bus_ids.size(); i++) {
          contingency.p_busid.push_back(bus_ids[i]);
          contingency.p_genid.push_back(gen_ids[i]);
          contingency.p_saveGenStatus.push_back(true);
        }
        // Add generator contingency to contingency list
        ret.push_back(contingency);
      }
    }
  }
  return ret;
}
```

Execute application.  argc and argv are standard runtime parameters

```
void gridpack::contingency_analysis::CADriver::execute(int argc, char** argv)
{
  // Create world communicator for entire simulation
  gridpack::parallel::Communicator world;

  // Get timer instance for timing entire calculation
  gridpack::utility::CoarseTimer *timer =
    gridpack::utility::CoarseTimer::instance();
  int t_total = timer->createCategory("Total Application");
  timer->start(t_total);

  // Read configuration file (user specified, otherwise assume that it is
  // call input.xml)
  gridpack::utility::Configuration *config
    = gridpack::utility::Configuration::configuration();
  if (argc >= 2 && argv[1] != NULL) {
    char inputfile[256];
    sprintf(inputfile,"%s",argv[1]);
    config->open(inputfile,world);
  } else {
```

```
  config->open("input.xml",world);
}

// Get size of group (communicator) that individual contingency calculations
// will run on and create a task communicator. Each process is part of only
// one task communicator, even though the world communicator is broken up into
// many task communicators
gridpack::utility::Configuration::CursorPtr cursor;
cursor = config->getCursor("Configuration.Contingency_analysis");
int grp_size;
double Vmin, Vmax;
if (!cursor->get("groupSize",&grp_size)) {
  grp_size = 1;
}
if (!cursor->get("minVoltage",&Vmin)) {
  Vmin = 0.9;
}
if (!cursor->get("maxVoltage",&Vmax)) {
  Vmax = 1.1;
}
gridpack::parallel::Communicator task_comm = world.divide(grp_size);

// Create powerflow applications on each task communicator
boost::shared_ptr<gridpack::powerflow::PFNetwork>
  pf_network(new gridpack::powerflow::PFNetwork(task_comm));
gridpack::powerflow::PFAppModule pf_app;
// Read in the network from an external file and partition it over the
// processors in the task communicator. This will read in power flow
// parameters from the Powerflow block in the input
pf_app.readNetwork(pf_network,config);
// Finish initializing the network
pf_app.initialize();
// Solve the base power flow calculation. This calculation is replicated on
// all task communicators
pf_app.solve();
// Some buses may violate the voltage limits in the base problem. Flag these
// buses to ignore voltage violations on them.
pf_app.ignoreVoltageViolations(Vmin,Vmax);
```

```
// Read in contingency file name
std::string contingencyfile;
if (!cursor->get("contingencyList",&contingencyfile)) {
  contingencyfile = "contingencies.xml";
}
// Open contingency file
bool ok = config->open(contingencyfile,world);

// Get a list of contingencies. Set cursor so that it points to the
// Contingencies block in the contingency file
cursor = config->getCursor(
    "ContingencyList.Contingency_analysis.Contingencies");
gridpack::utility::Configuration::ChildCursors contingencies;
if (cursor) cursor->children(contingencies);
std::vector<gridpack::powerflow::Contingency>
  events = getContingencies(contingencies);
// Contingencies are now available. Print out a list of contingencies from
// process 0 (the list is replicated on all processors)
if (world.rank() == 0) {
  int idx;
  for (idx = 0; idx < events.size(); idx++) {
    printf("Name: %s\n",events[idx].p_name.c_str());
    if (events[idx].p_type == Branch) {
      int nlines = events[idx].p_from.size();
      int j;
      for (j=0; j<nlines; j++) {
        printf(" Line: (from) %d (to) %d (line) \’%s\’\n",
            events[idx].p_from[j],events[idx].p_to[j],
            events[idx].p_ckt[j].c_str());
      }
    } else if (events[idx].p_type == Generator) {
      int nbus = events[idx].p_busid.size();
      int j;
      for (j=0; j<nbus; j++) {
        printf(" Generator: (bus) %d (generator ID) \’%s\’\n",
            events[idx].p_busid[j],events[idx].p_genid[j].c_str());
      }
    }
  }
}

// Set up task manager on the world communicator. The number of tasks is
// equal to the number of contingencies
gridpack::parallel::TaskManager taskmgr(world);
int ntasks = events.size();
taskmgr.set(ntasks);

// Evaluate contingencies using the task manager
int task_id;
char sbuf[128];
// nextTask returns the same task_id on all processors in task_comm. When the
// calculation runs out of task, nextTask will return false.
while (taskmgr.nextTask(task_comm, &task_id)) {
  printf("Executing task %d on process %d\n",task_id,world.rank());
  sprintf(sbuf,"%s.out",events[task_id].p_name.c_str());
  // Open a new file, based on the contingency name, to store results from
  // this particular contingency calculation
  pf_app.open(sbuf);
  // Write out information to the top of the output file providing some
  // information on the contingency
  sprintf(sbuf,"\nRunning task on %d processes\n",task_comm.size());
  pf_app.writeHeader(sbuf);
  if (events[task_id].p_type == Branch) {
    int nlines = events[task_id].p_from.size();
    int j;
    for (j=0; j<nlines; j++) {
      sprintf(sbuf," Line: (from) %d (to) %d (line) \'%s\'\n",
          events[task_id].p_from[j],events[task_id].p_to[j],
          events[task_id].p_ckt[j].c_str());
    }
  } else if (events[task_id].p_type == Generator) {
    int nbus = events[task_id].p_busid.size();
    int j;
    for (j=0; j<nbus; j++) {
      sprintf(sbuf," Generator: (bus) %d (generator ID) \'%s\'\
n",
      events[task_id].p_busid[j],events[task_id].p_genid[j].c_str());
   }
  }
  pf_app.writeHeader(sbuf);
  // Reset all voltages back to their original values
  pf_app.resetVoltages();
  // Set contingency
  pf_app.setContingency(events[task_id]);
  // Solve power flow equations for this system
  if (pf_app.solve()) {
   // If power flow solution is successful, write out voltages and currents
   pf_app.write();
   // Check for violations
   bool ok = pf_app.checkVoltageViolations(Vmin,Vmax);
   ok = ok & pf_app.checkLineOverloadViolations();
   // Include results of violation checks in output
   if (ok) {
    sprintf(sbuf,"\nNo violation for contingency %s\n",
      events[task_id].p_name.c_str());
   } else {
    sprintf(sbuf,"\nViolation for contingency %s\n",
      events[task_id].p_name.c_str());
   }
   pf_app.print(sbuf);
  }
  // Return network to its original base case state
  pf_app.unSetContingency(events[task_id]);
  // Close output file for this contingency
  pf_app.close();
 }
 // Print statistics from task manager describing the number of tasks performed
 // per processor
 taskmgr.printStats();

 timer->stop(t_total);
 // If all processors executed at least one task, then print out timing
 // statistics (this printout does not work if some processors do not define
 // all timing variables)
 if (contingencies.size()*grp_size >= world.size()) {
  timer->dump();
 }
}
```