Progress on the Second-Generation NTCC Physics Server Code

J. C. Wiley (U. Texas), H. St. John (GA),
R. H. Cohen, T. B. Yang (LLNL),
J. Kinsey (Lehigh), and NTCC team

Building on many of the ideas explored in the first year, a new NTCC physics server component is being developed. The framework for this server is primarily written in C++ with interfaces to Fortran modules from the module library. The server is part of a layered software architecture beginning with a client-server model for an abstract application. The server application has the ability to describe its constituents at runtime and to allow components to be dynamically edited. The server interface is comprised of a few general commands and data structures built on CORBA.
which provides both language and network protocol independence. The physics application component is derived from this generic application structure. The server is designed to accommodate both casual and expert users. The requirements of dynamic discovery, runtime configuration, and isolation of physics modules have driven the design.
Physics Server Code

One component of the NTCC demo suite. Solves system of transport equations obtaining needed sources and geometric coefficients from dataserver component and supplies state and profile information to client through a network CORBA interface. The server is designed to be externally configurable at runtime and to allow new physics modules and solvers to be installed at compile time. Provides a framework for a variable number of equations, grid points, transport models, etc. It can be run either interactively or as a batch process. Currently coded in C++ although there are natural stubs for including FORTRAN modules and for driving the framework with a scripted language. The internal structure of the code is not scripted.
C++ Physics Server

- Component of NTCC demo project responsible for advancing transport equations.

- "Transport Model Tester" - advances ion & electron temperatures with other fields obtained from database.

- Obtains density profiles, energy sources and sinks, geometry coefficients from the data server.

- Supplies configuration data and calculation results to client and receives configuration and control commands from client.
Requirements

- Easy for novice users - minimum learning curve (GUI)

- User configurable through interface.

- Internally organized to allow easy addition of new physics capabilities.

- Structured to leverage software components - internal and external reuse.

- Demonstrate “advanced” software tools - something beyond F77.

- Remote execution - client-server network architecture
Features of C++ Code

Incorporates ideas and experience gained from both Pyntd and earlier codes. Requirements for remote runtime configuration impose significant design constraints.

- Single language - C++ with CORBA (language neutral) network interface.

- Can solve systems with arbitrary number of equations, basis, grid, and can select transport models - all determined at runtime.

- All object parameters are edit-able through interface.

- New physics or numerical capabilities can be added through "plug-in models".
Main Generic Components
Structure dictated by requirements.

Universal (Reusable) Application Framework

- Network Interface - required by client-server model. Uses CORBA.

- Application Interface - presents controls and configuration data to network

- PropertyData - General data structure for passing complex and arbitrary collections of named parameters between components.

- Parts System - Allows dynamic runtime configuration.
Parts system

Developed to satisfy requirement of external configuration. For dynamic configuration, must be able to report types of objects which can be created before any parts exist.

- Abstract factory system

- Parts - base class of externally visible components.

- Registry - database of active parts. Allows self-discovery and runtime linking.
Each part implements three methods:

- `getClassName` - returns `propertyData` describing all externally known parameters and their default values. Static method so that it can be invoked before object created. Allows client to know what can be created.

- `getPartDescription` - returns `propertyData` describing current state of part. Also used to save state for restarts.

- `setPartDescription` - accepts `propertyData` setting some or all of parameters. All parts can be edited from interface.
Transport Components
Specializations for transport problem

- Fields - represents functions on time-space slab.

- FEB - finite element basis.

- Field Iterators

- Functions
Field Life Cycle

Since parts (fields) are created and linked at run time, they must proceed through a series of steps to discover and link with other dependent parts.

- create - each has a no argument constructor

- define - values assigned to state variables

- configure - part makes connections to other parts -

- initialize - brings part to valid initial state,

- detach - returns part to preconfiguration state, without changing value.
Field Evolution

Once all the parts for a problem are created, linked, and initialized the problem can be integrated in time. Each field has a number of methods used to advance the solution.

- iter - make one iteration update of the solution at $t + \delta t$.

- step - advance solution to next time interval.

- reset - return field to initial state.
Fields

- Represent a function over domain 
  \[ \{t, \delta t\} \times \{\rho \in \{0, 1\}\} \] by
  \[ u(r, t) = \sum_i (a_i(t)\phi(r)^i) \]

- Field represents function over
  \[ a(t)^k = a^k_0 + \frac{t - t_0}{\Delta t} \delta a^k. \]

- Main task of a field is to compute \( \delta a^k \)

- This is abstracted into the \textit{Iter} method of the field.

- Fields are classified as to how the \textit{Iter} method is implemented
FEB - Finite Element Basis

Functions are represented in space in terms of piecewise polynomials on a given grid. Each field has its own basis which may or may not be the same as other fields. Access to the finite element basis are through calls which in general do not depend on the type of basis, thereby decoupling the evolution logic with the internal representation. The exception is in fields that include equation solvers which can be greatly optimized for a particular element. However, this dependence is only local.

- Piecewise constant
- Piecewise linear
- Piecewise cubic
- Piecewise B-splines (any order)
The transport model tester uses solvers for a set of equations of the "H" form:

\[
H_1 \frac{\partial u}{\partial t} - \frac{\partial}{\partial x} \left( H_2 \frac{\partial u}{\partial x} + H_3 u + H_4 \right) - H_5 \frac{\partial u}{\partial x} - H_6 u - H_7 - H_8 (u - u^*) = 0
\]

which is just one example of more general of a more general evolution equation.

\[
A \frac{\partial u}{\partial t} - L(u) = S
\]

The only dependence in the code on the form of the equation is in the models which define the H equation and coefficients. H equations are only explicitly coupled through the drag term. Other couplings are handled by iteration.
Where for the coupled electron-ion energy equations we have:

\[ H_1 = \frac{3}{2} \psi_m \left( \frac{V_\rho}{\psi_m} \right)^{-\gamma+1} \]

\[ H_2 = V_\rho^{-\gamma+1} \psi_m \gamma \langle |\nabla \rho| \chi \rangle \]

\[ H_3 = V_\rho^{-\gamma+1} \psi_m \langle |\nabla \rho| \left( -|\nabla \rho| \chi \left( \frac{1}{n} \frac{\partial n}{\partial \rho} + \frac{\gamma}{V_\rho} \frac{\partial V_\rho}{\partial \rho} \right) + u_q + \frac{5}{2} \tilde{u}_\rho \right) \rangle \]

\[ H_4 = V_\rho \langle |\nabla \rho| q_{hp0} \rangle \]

\[ H_5 = -\frac{3}{2} \nu_m \dot{\psi}_m \rho \]

\[ H_6 = 0 \]

\[ H_7 = 0 \]

\[ H_8 = \psi_m^{\gamma} V_\rho^{-\gamma+1} \langle \nu_{ei} \rangle \]
Proxy window
Configure Window
Edit Window
2D Plot window
Menu for plot arrangement
Tiled Window
Using MPI In a Mixed C++/Fortran Environment

- The NTCC C++ and Python demo codes rely heavily on using existing Fortran modules from the module library (ftp://ftp.pppl.gov/pub/NTCC/). Many of these Fortran modules have embarassingly parallel sections that the NTCC codes will take advantage of by using the Message Passing Interface (MPI) method for parallelization.

- The question arises as to whether or not the public domain version of MPI, MPICH, will function in this environment. We have not found any previous reference to doing such mixed, parallelized, C++/Fortran computations.

- We have succeeded in developing a method of incorporating MPICH which currently works on LINUX platforms using KCC,PGCC, and EGCS C++ compilers and using PGF90 for the (parallel) Fortran sections.
A Simple Demonstration

- The Fortran modules which are parallelized must include the mpif.h file and all parallel directives not associated with MPI initialization and termination. These modules are compiled with PGF90 and archived in a static library (dynamic linking combined with mpi does not work for us at present).

- The MPI initialization \((\text{MPI} \text{. Init}(\text{argc, argv}))\) and final termination \((\text{MPI} \text{. Finalize()})\) must be done in function main on the C++ side. The C declarations file mpi.h is also included here.

- Discovering the required libraries and proper linking sequence is problematic and compiler dependent. For the Portland Group compiler we found that the appropriate linkage requires the following libraries:
  
  -lpmpich -lm -lpgf90 -lpgf90_rpm1 -lpgf902
  -lpgf90rtl -lm -lpgc -lgcc -lc -lgcc /usr/lib/crtm.o
Example C++ Main

/** driver for testing the c++/fpi coupling **/ 

#include "mpi.h"

#include <iostream>
using namespace std;
extern "C" void pghpf_init(int *);
extern "C" void fpi_(void);

int main (int argc, char* argv[]) {
    int numprocs,myid,nameLEN;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int zz = 0;
    pghpf_init(&zz);

    MPI_Init(&argc,&argv);

    fpi_();
    MPI_Finalize();

    exit(0); }

Parallel Fortran Segment

subroutine fpi

include 'mpif.h'

...... problem setup not shown

call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )

...... problem setup not shown

call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

...... the parallel loop:
sum = 0.0d0
do 20 i = myid+1, n, numprocs
   print *,"myid =",myid
   x = h * (dble(i) - 0.5d0)
   sum = sum + f(x)
20 continue
mypi = h * sum

collect all the partial sums

call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE PRECISION, & MPI_SUM,0, MPI_COMM_WORLD,ierr)