CHIMPS: A High-Performance Scalable Module for Multi-Physics Simulations


Center for Integrated Turbulence Simulations, Stanford University, Stanford, CA 94305, U.S.A.

As computational methods attempt to simulate ever more complex physical systems the need to couple independently-developed numerical models and solvers arises. This often results from the requirement to use different physical or numerical models for various portions of the domain of interest. In many situations it is also common to use different physical models that interact within the same domain of interest. The interaction between these models normally requires an exchange of information between the participating solvers. When the solvers that exchange information are distributed over a large number of processors in a parallel computer, the problem of exchanging information in an efficient and scalable fashion becomes complicated. This paper describes our efforts to develop a Coupler for High-Performance Integrated Multi-Physics Simulations, CHIMPS, that can enable the exchange of information between solvers and that automates the search, interpolation and communication processes in order to allow the developer to focus on appropriate strategies to couple solvers in an accurate and stable fashion. Our basic approach, the underlying technology and a number of examples are presented. A series of appendices are included with actual sample code and a description of the full CHIMPS API at the time of writing.

I. INTRODUCTION

Although in the last three decades there has been an explosive growth in the use of computational tools, the process of developing scientific software has not changed considerably. This process typically follows three phases: the first step is the definition of the required capabilities; next a variety of mathematical and physical models are identified and, then, a single computational code is written to achieve all objectives. In this approach, additional features can only be added if they are compatible with the overall structure of the software/algorithms already developed. In addition, updates and modernization of the code structure typically require a complete rewrite and, therefore, a considerable investment. Moreover, portions of the code are considered to be legacy and little or no expertise is available to update and maintain the code. An alternative approach is to build flexible computational infrastructures that consist of several independent but integrated solvers. Each component performs a specific task and addresses a specific physical aspect of the problem; the ability to integrate allows for the timely and accurate exchange of information and ensures that new features or updated models can be included without disrupting the entire infrastructure and with reasonable effort. An additional benefit of this strategy is that various existing component modules or solvers can be rapidly combined to solve new problems without the tremendous overhead needed to create an environment from scratch.

Large multidisciplinary problems require the collaboration of a large group of scientists and the development of an extensive simulation environment. Within the Department of Energy Advanced Simulation and Computing (ASC) program, the Center for Integrated Turbulence Simulations (CITS) at Stanford University is performing simulations of the flow through an entire jet engine. The radically different physics, length scales and dynamics of the problem prompt the use of different simulation technology. In the compressor

*Authors are listed in alphabetical order.
and turbine the flow is transonic, includes a very large number of moving parts and the geometric configuration is rather simple. On the other hand, in the combustor, the low speed air, is mixed with liquid fuel and reactions take place. A compressible Unsteady Reynolds-Averaged Navier-Stokes (URANS), structured multiblock code is sufficient to capture the flow features in the turbomachinery components, whereas an unstructured mesh Large-Eddy Simulation (LES), reactive flow algorithm is required in the combustor to reproduce the intricate injector passages and to account for the proper levels of mixing and cooling flows. In addition, the modeling of the liquid fuel atomization requires the detailed computation of the evolution of liquid sheets and their interaction with the gas phase. Finally, if aero-structural phenomena are to be accounted for, it is necessary to couple the fluid and structural solvers through an appropriate fluid-structure interface.

The interaction between all of these solvers is usually accomplished via communication of information computed by each solver. This interaction can be based on artificially-imposed boundary conditions or simply on information interpolated between the grids that each solver computes on. Although the problem of interpolation may appear straightforward, it is in fact complicated by two fundamental issues. Firstly, in modern computing environments, the participating codes are usually domain decomposed into a large number of processors. The data to be interpolated can reside anywhere in the distributed memory parallel computer. The search process that identifies the mesh cells that provide information to a given solver (and the processor that they reside on) and the ensuing exchange of information have often been described as the $M \times N$ problem in the literature. The number of processors that are typically involved in these search, interpolation, and communication patterns can be very large. The efficient and scalable (both in CPU time and in used memory) solution of the $M \times N$ problem is not straightforward. Secondly, although linear (bi-linear, tri-linear) interpolation is relatively straightforward to implement, it is unable to guarantee the accuracy, conservativeness, and stability of the coupled solution except in the limit of an infinitely fine mesh: the order of accuracy of a coupled solution can drop to first order if care is not taken to ensure accuracy.

The current implementation of CHIMPS addresses the first issue: the solution of the $M \times N$ problem using tri-linear mappings for the solution inside each element of a mesh. At the moment, CHIMPS does not provide any assistance with the second issue (it is left entirely to the user/developer) although future efforts will focus on addressing this issue within the context of the CHIMPS API.

This paper is organized to address the following objectives:

- Provide an overview of the functionality and performance of CHIMPS for some representative problems.
- Describe the full CHIMPS API and the underlying technology choices we have made.
- Provide some representative test cases (including the full source code in the appendices) so that this paper may be used as a Users’ Manual for CHIMPS.

## II. CODE COUPLING ENVIRONMENT

The code coupling environment is designed to facilitate the setup and execution of large-scale integrated simulations. This is accomplished to the introduction of a comprehensive API and a series of guidelines or the development of integrated tools.

### A. General Framework

The integrated simulation environment consists of three components:

- the applications;
- the driver;
- the CHIMPS library.

The application codes are the basic building blocks providing the algorithms/physical models required to perform the simulations; the driver, on the other hand, is the controlling script, organizing the work-flow of the simulation and synchronizing the exchange of data between applications. The CHIMPS library, is the communication hub that enables scalable communication between multiple applications through the use of
an efficient parallel search engine. The driver script is the heart of the simulation; it can be programmed in Python or FORTRAN and performs a sequence of operations, logically divided in phases:

1. **initialization**: the parallel environment for the integrated simulation is defined, the application codes and the processors assigned to each application are identified.

2. **registration**: each application defined its computational domain, grid and associated data.

3. **interface set-up**: the data and the physical location of the exchanges required to perform the coupling between the applications are defined.

4. **run applications**: each component application perform one or multiple solutions steps.

5. **data exchange**: at the interfaces data are exchanged.

6. **finalize**.

The steps (3)-(6) might be repeated in typical simulations.

The application code are required to comply with a well defined API (see appendix) to participate in an integrated simulation. This allows the codes to register their physical domain and data, and to, and a Python/FORTRAN-based API. At its heart, CHIMPS performs operations (interpolation, integration) and information exchanges without the need for user/programmer involvement. The basic idea is to facilitate the setup and execution of large-scale integrated simulations so that engineers and scientists can focus on the details of the science and not the interpolation/communication. Moreover by creating a comprehensive API, codes that use CHIMPS can benefit from future improvements to the search, interpolation, and communication algorithms (all under the hood) without the need to modify their code. A coupled simulation using the CHIMPS framework requires that each participating application provide: its data through a number of code-specified field names. The geometrical location of its data. Currently CHIMPS supports meshes and points as geometrical entity. Also, the current mesh definition is through an unstructured format that supports the four basic elements primitives (tetrahedra, prisms, pyramids and hexahedrals). Any application can then request specific data from any other application running within the same integrated simulation. The request is formulated again in terms of data and a geometrical entity where data are required at. An interface is then defined by linking the geometrical entities that participate in the data exchange and by specifying what kind of operation is requested (at the moment only interpolation and integration are supported). This data exchange and manipulation is carried out throughout the parallel network. The CHIMPS search engine supports both surface (minimum distance) and volume (containment) searches for the requested data. In addition, it has a fail-safe mode that attempts a volume search procedure and, if the volume search is unsuccessful, attempts a minimum distance search. In order to perform an integrated simulation, the user must prepare a driving program (either a FORTRAN program or a Python script) and a set of application-specific subroutines if one of the participating applications does not directly contain a CHIMPS interface. Although CHIMPS is written as a general framework (without direct reference to any specific application) the driving program MUST have detailed knowledge of the coupling requirements and restrictions of the applications to be coupled.

B. **Definitions**

1. **Registration**
2. **Interface**
3. **Exchange**

C. **Numerical issues** (Ken, Magnus)

D. **Current capabilities**

III. TECHNICAL COMPONENTS

- The parallel environment (Edwin, Frank)
- The search algorithm (Edwin)
A. Interpolation vs. Integration in CHIMPS

Volume integration interfaces play an important role in finite volume solvers. By definition, a stored quantity \( \phi_i \) in a finite volume code represents a control-volume averaged quantity,

\[
\phi_i = \int_{V_i} \phi \, dx / V_i ,
\]

with \( V_i \) the cell control volume. This implies that if two codes are to be coupled that employ overlapping or intersecting grids, Eq. 1 must be solved to determine \( \phi_i \). For simplicity, it will be assumed in the following that two codes, code A and code B, are to be coupled, with code B consisting of a fine mesh (blue) that overlays parts of a code A coarse mesh (red), see left part of Fig. 1.

Let \( i \) be the code A cell in the lower right corner of Fig. 1. To calculate \( \phi_i \) from code B data \( \phi_j \) exactly, CHIMPS has to first identify all cells \( G \) of code B, that have any common volume with \( V_i \) (large green dots in center of Fig. 1), calculate the common volume \( V_{i,j} = V_i \cap V_j \) (shaded blue areas in center of Fig. 1), and then solve

\[
\phi_i = \sum_{j \in G} \phi_j V_{i,j} / V_i .
\]

This exact formulation will be implemented in a future version of CHIMPS. The current version employs a simpler, faster, approximate scheme that reverts back to the exact formulation in the case of exactly overlapping grids, i.e. \( V_{i,j} = V_i \) or \( V_{i,j} = 0 \). The right side of Fig. 1 shows the more general case of not exactly overlapping grids. In a first step, CHIMPS identifies all code B control volume centroids \( F \) that are within \( V_j \) (large green dots in right of Fig. 1) and then solves

\[
\phi_i = \sum_{j \in F} \phi_j V_i / \sum_{j \in F} V_j .
\]

To utilize the integration interface in a driver program, the mesh of code A has to be registered as a mesh entity and the cell centroids of code B have to be registered as a point entity. Furthermore, code B has to provide the cell control volumes \( V_j \) in the `codeB_getPointData` call. These then have to passed to CHIMPS in the `chimps_setPointData` with the variable name `CELL_VOLUME`. An example implementation of an integration interface in a driver program is listed in Appendix B.

The actual implementation of an integration interface in CHIMPS consists of the following steps:

1. generate ADT for code A’s mesh
2. send code B’s centroid coordinates \( x_j \) to code A’s processors in a balanced way

Figure 1. CHIMPS volume integration code B (blue grid) to code A (red grid), exact integration (center), approximate integration (right).
3. find code A’s mesh element $e(x_j)$ that contains $x_j$
4. send element information $e(x_j)$ back to code B’s processor that contains $x_j$
5. build list of unique code A elements $e'(x_j)$ on each code B processor
6. build communication structure for $e'(x_j)$
7. on code B’s processors calculate $\phi_{e'}(x_j) = \sum_{j \in e'(x_j)} \phi_j V_j$ and $V_{e'}(x_j) = \sum_{j \in e'(x_j)} V_j$
8. send $\phi_{e'}(x_j)$ and $V_{e'}(x_j)$ to code A processor containing $e'(x_j)$
9. on code A processors calculate $\phi_i = \sum_{np \text{ codeB}} \phi_{e'}(x_j) / \sum_{np \text{ codeB}} V_{e'}(x_j)$

Note, that step 1 can be skipped if code’s A mesh has not changed, and steps 2-6 can be skipped, if neither code A’s nor code B’s mesh has changed. Furthermore, the ADT build in step 1 is actually the same as for an interpolation interface from code A to code B. Thus, in the common case that code B requests interpolated data from code A, and code A requests integrated data from code B, the ADT needs to be build only once, thereby reducing computational time, see also Sec. C.

The above algorithm ensures memory scalability even in the extreme case that all code B centroids are within a single code A mesh element. Also, the dual-step integration, first on code B’s processors in step 7 and then on code A’s processors in step 9 ensures that actual parallel communication is kept to a minimum, thereby ensuring excellent scalability.

IV. SOFTWARE VERIFICATION

A. Analytical tests (interpolation, order of accuracy, element independency) (Frank, Marcus)
B. Simple problems (pipes, swirling pipes, etc.) (Goradz, Georgi)
C. Scalability

One of the crucial features of the CHIMPS library is it’s ability to scale up well even on massively parallel, distributed memory computer systems. In order to quantify the scalability, a simple interpolation coupling of two codes, code A and code B is performed, see Appendix A. Both codes exist on the same group of processors. In code A, $N$ hexahedral elements are defined in a unit cube, from which code B requests two interpolated scalar values at $M$ randomly distributed points inside the unit cube. Both codes are themselves perfectly load-balanced.

Figure 2 shows the speed-up as a function of the number of processors for the first updateInterface and any subsequent updateInterface call. Note that for the first updateInterface call the ADT and communicators are build, the data is exchanged and the interpolation is performed, whereas for any subsequent updateInterface call, only the latter two steps have to be performed. To obtain a meaningful scaling relation, the problem size, that is $N$ and $M$, is increased once the speed-up drops off due to an insufficient number of elements/points per processor. To insure compatibility, the speed-up of the larger problem is then scaled to the obtained speed-up of the smaller problem at the previous number of processors.

The observed speed-up for the initial updateInterface is slightly hyper-linear. This is due to the fact that the ADT build and search scales hyper-linear due to Edwin???. The subsequent calls to updateInterface also show excellent scalability, with a noticeable drop-off in speed-up once the problem size becomes too small. However, in practical applications this drop-off is not significant, since in terms of wall-clock time, the subsequent calls to updateInterface are significantly faster than the initial call. For example, in the $N = 536.9 \cdot 10^6$ and $M = 134.2 \cdot 10^6$ case on 256 processors the wall-clock execution time for the initial call to updateInterface is 50.5s compared to 1.9s for any one of the subsequent calls.

V. APPLICATIONS

A. The Jet Engine
B. The Rotorcraft
VI. REFERENCE


Figure 2. CHIMPS scalability for initial updateInterface call (left) and subsequent updateInterface calls (right) with $N = 16.8 \cdot 10^6$ and $M = 4.2 \cdot 10^6$ (red circles), $N = 134.2 \cdot 10^6$ and $M = 33.6 \cdot 10^6$ (green squares), and $N = 536.9 \cdot 10^6$ and $M = 134.2 \cdot 10^6$ (blue triangles).
VII. APPENDIX

A. Scalability driver program

program ScalabilityDriver
use chimps_m
use codeA_m
use codeB_m
implicit none

include 'mpif.h'

character(len=32) :: groupName, varNames(2), interfaceNames(1)
integer :: ierr, myrank, nprocs, groupComm, myGroupRank, iter
integer(IP) :: nNodes, nTet, nPyra, nPrism, nHex, nPoints, nVars, nInterfaces
real(RP), allocatable, dimension(:,:) :: xyz, data
integer(IP), allocatable, dimension(:,:) :: tetConn, pyraConn, prismConn, hexaConn

! for timing
real(RP), dimension(2) :: ts, dt, dtmean

! Initialize mpi...
call mpi_init(ierr)
call MPI_comm_rank(MPI_COMM_WORLD, myrank, ierr)
call MPI_comm_size(MPI_COMM_WORLD, nprocs, ierr)

dt = 0.0_RP

! initialize chimps on all processors using the same group name.
groupName = 'CODEA_CODEB'
call chimps_initialize(groupComm, groupName, MPI_COMM_WORLD)
call chimps_setParam('HIGH', 'VERBOSITY')
call codeA_init(groupComm)
call codeB_init(groupComm)
call MPI_comm_rank(groupComm, myGroupRank, ierr)

code A mesh registration
call codeA_getMeshSize(nNodes, nTet, nPyra, nPrism, nHex)
allocate(xyz(3, nNodes), tetConn(4, nTet), pyraConn(5, nPyra), prismConn(6, nPrism), hexaConn(8, nHex))
call codeA_getMeshGeom(xyz, hexaConn)
call chimps_setMeshGeom(xyz, nNodes, tetConn, nTet, pyraConn, nPyra, prismConn, nPrism, hexaConn, "CODEA_MESH")
deallocate(xyz, tetConn, pyraConn, prismConn, hexaConn)

! code B point registration
call codeB_getPointSize(nPoints)
allocate(xyz(3, nPoints))
call codeB_getPointGeom(xyz)
call chimps_setPointGeom(xyz, nPoints, "CODEB_POINTS")

d code B point registration

code B point registration
allocate(xyz(3, nPoints))
call codeB_getPointGeom(xyz)
call chimps_setPointGeom(xyz, nPoints, "CODEB_POINTS")

! register interpolation interface
call chimps_setInterface(\"MF\_INTERFACE\", \"CHIMPS\_INTERPOLATE\", \"CHIMPS\_INTER\", \"CHIMPS\_INTERFACE\", \"CODEA\_CODEB\", \"CODEB\_CODEA\")

! register requested variable names
varNames(1) = 'PHI'
vartNames(2) = 'PSI'
nVars = 2

call chimps_setPointRequest(varNames, nVars, 'CODEB_POINTS')

! MAIN LOOP
do iter = 1, 3

! register solution on mesh for code A
varNames(1) = 'PHI'
vartNames(2) = 'PSI'
nVars = 2
allocate(data(nVars, nNodes))
call codeA_getMeshData(data)
call chimps_setMeshData(data, varNames, nVars, 'CODEA_MESH')
deallocate(data)

! timing main work
ts(1) = MPI_WTIME()

! perform interpolation
interfaceNames(1) = 'MF\_INTERFACE'
allocate(data(nVars, nPoints))
call chimps_setInterface(data, varNames, nVars, 'CODEA_MESH')
deallocate(data)

! timing main work
ts(2) = MPI_WTIME()

if (iter == 1) then
  dt(1) = ts(2) - ts(1)
else
  dt(2) = dt(2) - ts(1)
end if

! retrieve interpolated data
varNames(1) = 'PHI'
varNames(2) = 'PSI'
nVars = 2
allocate(data(nVars, nPoints))
call chimps_getPointData(data, varNames, nVars, 'CODEB_POINTS')
call codeB_setPointData(data, varNames, nVars, 'CODEB_POINTS')
deallocate(data)
end do

! output of timing information
call MPI_REDUCE(dt(1), dtMean(1), 2, MPI_DOUBLE_PRECISION, MPI\_SUM, 0, MPI\_COMM\_WORLD, ierr)

if (myrank == 0) then
  dtMean = dtMean/real(nprocs, RP)
dtMean = dtMean/2.0
write (*,*) 'Timing results: '
write (*,*) 'Chimps last updateInterface: ', dtMean
end if
write (*,'(a,1x,4f12.5)') ' Chimps 2-n avg update Interface: ',dtMean(2)
end if

call chimps_finalize()
call codeA_finalize()
call codeB_finalize()
call mpi_finalize(ierr)
end program ScalabilityDriver

B. Integration interface driver

program IntegrationDriver
use chimps_m
use codeA_m
use codeB_m
implicit none
include 'mpif.h'

character(len=32) :: groupName, varNames(3), interfaceNames(1)
integer :: ierr, myrank, nprocs, groupComm, myGroupRank, iter
integer(IP) :: nNodes, nTet, nPyra, nPrism, nHex, nPoints, nVars, nInterfaces
real(RP), allocatable, dimension(:,:) :: xyz, data
integer(IP), allocatable, dimension(:,:) :: tetConn, pyraConn, prismConn, hexaConn

! Initialize mpi...
call mpi_init(ierr)
call MPI_comm_rank(MPI_COMM_WORLD, myrank, ierr)
call MPI_comm_size(MPI_COMM_WORLD, nprocs, ierr)
! initialize chimps on all processors using the same group name.
groupName = 'CODEA_CODEB'
call chimps_initialize(groupComm,groupName,MPI_COMM_WORLD)
call chimps_setParam('HIGH','VERBOSITY')
call codeA_init(groupComm)
call codeB_init(groupComm)
call MPI_comm_rank(groupComm, myGroupRank, ierr)
! code A mesh registration
call codeA_getMeshSize(nNodes,nTet,nPyra,nPrism,nHex)
allocate(xyz(3,nNodes),tetConn(4,nTet),pyraConn(5,nPyra),prismConn(6,nPrism),hexaConn(8,nHex))
call codeA_getMeshGeom(xyz,hexaConn)
call chimps_setMeshGeom(xyz,nNodes,tetConn,nTet,pyraConn,nPyra,prismConn,nPrism,hexaConn,nHex,groupName)
deallocate(xyz,tetConn,pyraConn,prismConn,hexaConn)
! code B mesh registration as point entity

call codeB_getPointSize(nPoints)
allocate(xyz(3,nPoints))
call codeB_getPointGeom(xyz)
call chimps_setPointGeom(xyz,nPoints,groupName)
! register interpolation interface

call chimps_setInterface("MY_INTERFACE",CHIMPS_INTERPOLATE, &
   "CODEA_MESH", "CODEA_CODEB", & ! dest
   "CODEB_POINTS", "CODEA_CODEB") ! src
! register requested variable names

varNames(1) = 'PHI'
varNames(2) = 'PSI'
nVars = 2
allocate(data(nVars,nNodes))
call codeB_getPointData(data)
call chimps_setPointData(data,varNames,nVars,groupName)
deallocate(data)

! timing main work

tst(1) = MPI_WTIME()
! perform interpolation

interfaceNames(1) = "MY_INTERFACE"
interfaces = 1
call chimps_updateInterface(interfaceNames,interfaces)
! retrieves integrated data

varNames(1) = 'PHI'
varNames(2) = 'PSI'
nVars = 2
allocate(data(nVars,nPoints))
call chimps_getMeshData(data,varNames,nVars,groupName)
call codeA_setPointData(data,varNames,nVars,groupName)
deallocate(data)
end do

call chimps_finalize()
call codeA_finalize()
call codeB_finalize()
call mpi_finalize(ierr)
end program IntegrationDriver
C. Pitching Airfoil Test Case

1. Overall structure of a driver program

The final executable to run a multi-code integrated simulation is constructed by writing and compiling a driver program, which is the most outer layer for the entire set of programs. All the solver and CHIMPS modules are loaded and API routines are actually called in it.

Figure ?? shows the typical structure of a driver program which would be common to most applications. The sequence of operations in a driver is as follows:

1–2. As is common to all MPI applications, the first step is the MPI initialization, which is followed by the initialization of CHIMPS and each solver. Note that, along with the global communicator for driver and CHIMPS, a local communicator for each solver should be also defined here. In CHIMPS, it is actually handled by `chimps_initialize` which returns a local communicator as an output, and hence it is not necessary to call `mpi_comm_split` separately within the driver.

3. After initialization, the first thing to do is to transfer information on all the involved geometric objects (such as interface points or volume meshes) from each solver and register them to CHIMPS, which is done by calling `solver_getPoint(Mesh)Size`, `solver_getPoint(Mesh)Geom` and `chimps_setPoint(Mesh)Geom` in this order.

4. Once all the points and volume meshes are registered to CHIMPS, CHIMPS should construct interfaces, which are the relationships between named point and mesh objects, via `chimps_setInterface`.

5. Now the list of flow variables requested by each solver for its interfaces should be registered to CHIMPS for later interpolations, which is done by `chimps_setPointRequest`. Note that CHIMPS uses CGNS names to identify flow variables.

6. The list of all the relevant flow variables and actual solution data at the entire mesh points are transferred from each solver to CHIMPS by `solver_getMeshData` and `chimps_setMeshData`. Here, it is users’ responsibility to guarantee that all the participating solvers provide proper flow variables to CHIMPS in order to prevent the interpolation failure.

7. Given all the transferred data and interfaces constructed, CHIMPS performs searches and interpolations via `chimps_updateInterface`.

8. With searches and interpolations completed, the interpolated data are transferred from CHIMPS back to each solver by `chimps_getPointData` and `solver_setPointData`. Within each solver, these data are employed as a boundary condition.

9. Given the interface data, each solver performs a time marching.

10. If the final time step is reached, each solver and CHIMPS are finalized. Otherwise, repeat 6–9 in case that all the relevant geometric objects are stationary. If the problem involves moving grids in time, repeat 3–9 to reflect changes in geometry at every time step.

Although CHIMPS is entirely written in Fortran90, a driver can be written in other programming languages in principle, more preferably in script languages like Python or Perl. Currently, CHIMPS supports both Fortran- and Python-based APIs. Note that f2py provides an efficient environment which enables all the Fortran-based API routines accessible from Python without any necessity of rewriting it.

2. Example of a driver program

As an example, we provide an actual driver program in this section, which was used for the coupled simulation of dynamic stall, so that users can utilize it as a template in writing their own ones. For this specific problem, the following names are prescribed to identify geometric objects and interfaces:

(1) Names internally defined within each solver
- These internal names should be used as an argument of solver API routines.
  • SUmbInterfaceFamily: Name for SUmb interface points used within SUmb.
  • SUmbVolume: Name for SUmb volume meshes (coordinates and connectivity) used within SUmb.
  • CDPInterface: Name for CDP interface points used within CDP.
  • CDPInterface: Name for CDP interface points used within CDP.
  • default-interior: Name for CDP volume meshes (coordinates and connectivity) named within CDP.

(2) Names newly created during registration to CHIMPS
- These names should be used as an argument of CHIMPS API routines.
  • SUmb_POINTS: Name registered to CHIMPS for SUmb interface points.
  - Users can also register the same names as in (1) to prevent the confusion.
  • SUmb_POINTS: Name registered to CHIMPS for SUmb interface points.
CDP_POINTS: Name registered to CHIMPS for CDP interface points.

SUMB_MESH: Name registered to CHIMPS for SUmb volume meshes.

CDP_MESH: Name registered to CHIMPS for CDP volume meshes.

SUMB_INTERFACE: Name registered to CHIMPS for the SUmb interface.

CDP_INTERFACE: Name registered to CHIMPS for the CDP interface.

Also note that the assignment of constant free-stream density is manipulated within the driver program, instead of directly transferring it from CDP via CHIMPS. For minute details, the code is annotated in red below.

```
! Driver to couple CDP and SUmb for flow over a pitching airfoil.
! - SUmb covers the near-blade domain pitching with the airfoil.
! - CDP covers the stationary outer domain.
!
program coupled_pitching_xf
use mpi
use cdp_if_coupler_m
This module contains all the CDP API routines.
use sumb_coupler_m
This module contains all the SUmb API routines.
use chimps_m
This module contains all the CHIMPS API routines.
implicit none

! ********** Beginning of driver parameters **********
! Variables needed for this driver are specified
! as parameters here...
integer(IP), parameter :: nSteps = 1000, &
Number of unsteady time steps for this session.
nProcSUmb = 8, &
Number of processors alloted to SUmb. The rest is for CDP.
ntwittStepsVolSUmb = 2, &
Writing interval for SUmb volume flow fields.
ntwittStepsSurSUmb = 2
Writing interval for SUmb surface flow fields.
logical, parameter :: movGridSUmb = .true., &
Whether SUmb grids are moving or not.
movGridCDP = .false., &
Whether CDP grids are moving or not.
adjustDt = .false.
Whether the driver readjusts the unsteady time step of each solver.

! ********** End of driver parameters **********

integer(IP), allocatable :: tetraConn(:,:), pyraConn(:,:), &
prismConn(:,:), hexaConn(:,:) 
real(RP) :: tmpDrv1, tmpDrv2, rhoConstDrv, dtSUmb, dtCDP 
real(RP), allocatable :: xyz(:,:), data(:,:)
character(len=80) :: groupName
character(len=80), allocatable :: varNames(:), interfaceNames(:)
character(len=4) :: ntstStr
character(len=2) :: myIDStr

! ********** Beginning of the program **********
!
! Initialize mpi...
call mpi_init(ierr)
call mpi_comm_size(mpi_comm_world, nProcsDrv, ierr)
call mpi_comm_rank(mpi_comm_world, myIDDrv, ierr)
!
! Define groupName...
if(myIDDrv < nProcsSUmb) groupName = 'SUMB'
if(myIDDrv >= nProcsSUmb) groupName = 'CDP'
!
! Initialize CHIMPS...
call chimps_initialize(groupComm, groupName, mpi_comm_world)
chimps
initialize
returns
groupComm
which is a local communicator for each solver.
!
! Initialize participating solvers...
if(groupName == 'SUMB') then
  call sumb_initialize('param.in', groupComm)
else if(groupName == 'CDP') then 
call cdp_initialize('cdp_if.in', groupComm)
dendif
!
! Get some parameters from SUmb for later use...
tmpDrv1 = 0.
if(groupName == 'SUMB') then
call sumb_getParam(tmpDrv1, 'density for initialization')
density for initialization
is the SUmb keyword for free-stream density.
Its value is obtained via sumb
getParam.
dendif
call mpi_allreduce(tmpDrv1, rhoConstDrv, 1, mpi_real8, mpi_max, &
mpi_comm_world, ierr)
The obtained value is shared by all processors.
!
! Get some parameters from CDP for later use...
tmpDrv2 = 0.
if(groupName == 'SUMB') then
call cdp_getParam(tmpDrv2, 'density for initialization')
density for initialization
is the CDP keyword for free-stream density.
Its value is obtained via cdp
getParam.
dendif
```

10 of 14

American Institute of Aeronautics and Astronautics
deallocate(xyz)
! Setup the SUmb mesh stuffs...
call sumb_getMeshSize(nNodes, nTetra, nPyra, nPrism, nHexa, & 'SUmbAirfoil')
allocate(xyz(3,nNodes), tetraConn(4,nTetra), pyraConn(5,nPyra), &
        prismConn(6,nPrism), hexaConn(8,nHexa))
call sumb_getMeshGeom(xyz, tetraConn, pyraConn, prismConn, &
        hexaConn, 'SUmbAirfoil')
call chimps_setMeshGeom(xyz, nNodes, tetraConn, nTetra, &
        pyraConn, nPyra, prismConn, nPrism, &
        hexaConn, nHexa, 'SUmb_MESH')
deallocate(xyz, tetraConn, pyraConn, prismConn, hexaConn)
else if(groupName == 'CDP') then
! Setup the point stuffs where data will be requested by CDP...
call cdp_getPointSize(nPoints, 'cdp_interface')
allocate(xyz(3,nPoints))
call cdp_getPointGeom(xyz, 'cdp_interface')
call chimps_setPointGeom(xyz, nPoints, 'CDP_POINTS')
deallocate(xyz)
! Setup the CDP mesh stuffs...
call cdp_getMeshSize(nNodes, nTetra, nPyra, nPrism, nHexa, &
        'default-interior')
allocate(xyz(3,nNodes), tetraConn(4,nTetra), pyraConn(5,nPyra), &
        prismConn(6,nPrism), hexaConn(8,nHexa))
call cdp_getMeshGeom(xyz, tetraConn, pyraConn, prismConn, &
        hexaConn, 'default-interior')
call chimps_setMeshGeom(xyz, nNodes, tetraConn, nTetra, &
        pyraConn, nPyra, prismConn, nPrism, &
        hexaConn, nHexa, 'CDP_MESH')
deallocate(xyz, tetraConn, pyraConn, prismConn, hexaConn)
endif
! Setup the interfaces which are relationships between named point- & mesh-stuffs...
! This should be called by all participating processors...
call chimps_setInterface('SUMB_INTERFACE', &
        CHIMPS_INTERPOLATE_FAILSAFE, &
        'SUmb_MESH', 'SUmb', &
        'SUMB_POINTS', 'SUmb')
call chimps_setInterface('CDP_INTERFACE', &
        CHIMPS_INTERPOLATE_FAILSAFE, &
        'CDP_MESH', 'CDP', &
        'CDP_POINTS', 'CDP')

! Main iteration loop.
iter_loop: do n=1,nSteps
! Solvers determine the flow variables they need & or should provide from their local domains...
if(groupName == 'SUMB') then
! Data SUmb requests...
  nVars = 7
  allocate(varNames(nVars))
  CGNS names are used to identify flow variables.
  varNames(1) = 'VelocityX'
  varNames(2) = 'VelocityY'
  varNames(3) = 'VelocityZ'
  varNames(4) = 'TurbulentEnergyKinetic'
  varNames(5) = 'TurbulentDissipation'
  varNames(6) = 'TurbulentScalarV2'
  varNames(7) = 'TurbulentScalarF'
  call chimps_setPointRequest(varNames, nVars, 'SUMB_POINTS')
deallocate(varNames)
! Data SUmb should provide...
  nVars = 7
  allocate(varNames(nVars))
  varNames(1) = 'VelocityX'
  varNames(2) = 'VelocityY'
  varNames(3) = 'VelocityZ'
  varNames(4) = 'TurbulentEnergyKinetic'
  varNames(5) = 'TurbulentDissipation'
  varNames(6) = 'TurbulentScalarV2'
  varNames(7) = 'TurbulentScalarF'
  allocate(data(nVars,nNodes))
call sumb_getMeshData(data, varNames, nVars, 'SUmbAirfoil')
call chimps_setMeshData(data, varNames, nVars, 'SUMB_MESH')

deallocate(varNames, data)

else if(groupName == 'CDP') then
  ! Data CDP requests...
  nVars = 7
  allocate(varNames(nVars))
  varNames(1) = 'VelocityX'
  varNames(2) = 'VelocityY'
  varNames(3) = 'VelocityZ'
  varNames(4) = 'TurbulentEnergyKinetic'
  varNames(5) = 'TurbulentDissipation'
  varNames(6) = 'TurbulentScalarV2'
  varNames(7) = 'TurbulentScalarF'
  call chimps_setPointRequest(varNames, nVars, 'CDP_POINTS')

deallocate(varNames)

  ! Data CDP should provide...
  nVars = 7
  allocate(varNames(nVars))
  varNames(1) = 'VelocityX'
  varNames(2) = 'VelocityY'
  varNames(3) = 'VelocityZ'
  varNames(4) = 'TurbulentEnergyKinetic'
  varNames(5) = 'TurbulentDissipation'
  varNames(6) = 'TurbulentScalarV2'
  varNames(7) = 'TurbulentScalarF'
  allocate(data(nVars,nNodes))
  call cdp_getMeshData(data, varNames, nVars, 'default-interior')
  call chimps_setMeshData(data, varNames, nVars, 'CDP_MESH')

deallocate(varNames, data)
endif

! Do the actual exchange...  

nInterfaces = 2
allocate(interfaceNames(nInterfaces))
interfaceNames(1) = 'SUMB_INTERFACE'
interfaceNames(2) = 'CDP_INTERFACE'

call chimps_updateInterface(interfaceNames, nInterfaces)

deallocate(interfaceNames)

! Get the data from CHIMPS and provide them to the appropriate solver...
if(groupName == 'SUMB') then
  nVars = 7
  ! A little bit of trick to provide the constant density to SUMb...
  allocate(varNames(nVars+1))
  varNames(8) is reserved for density.
  varNames(1) = 'VelocityX'
  varNames(2) = 'VelocityY'
  varNames(3) = 'VelocityZ'
  varNames(4) = 'TurbulentEnergyKinetic'
  varNames(5) = 'TurbulentDissipation'
  varNames(6) = 'TurbulentScalarV2'
  varNames(7) = 'TurbulentScalarF'
  allocate(data(nVars+1,nPoints))
  data(8,:) is reserved for density.

  call chimps_getPointData(data, varNames, nVars, 'SUMB_POINTS')

  A constant density is directly specified here.
  varNames(nVars+1) = 'Density'
  data(nVars+1,:) = rhoConstDrv

  call sumb_setPointData(data, varNames, nVars+1, 'SUmbRhouvwInterfaceFamily')

deallocate(varNames, data)
else if(groupName == 'CDP') then
  nVars = 7
  allocate(varNames(nVars))
  varNames(1) = 'VelocityX'
  varNames(2) = 'VelocityY'
  varNames(3) = 'VelocityZ'
  varNames(4) = 'TurbulentEnergyKinetic'
  varNames(5) = 'TurbulentDissipation'
  varNames(6) = 'TurbulentScalarV2'
  varNames(7) = 'TurbulentScalarF'
  allocate(data(nVars,nPoints))

  call chimps_getPointData(data, varNames, nVars, 'CDP_POINTS')
call cdp_setPointData(data, varNames, nVars, 'cdp_interface')
deallocate(varNames, data)
endif

! Determine the unsteady time step...
if(adjustDt .and. n == 1) then
tmpDrv1 = 0.
tmpDrv2 = 0.
endif

Time-step sizes for the two solvers are obtained via sumb_getParam:
if(groupName == 'SUMB') then
call sumb_getParam(tmpDrv1, 'unsteady time step (in sec)')
else if(groupName == 'CDP') then
call cdp_getParam(tmpDrv2, 'DT')
endif

Obtained values are shared by all processors:
call mpi_allreduce(tmpDrv1, dtSUmb, 1, mpi_real8, mpi_max, &
                   mpi_comm_world, ierr)
call mpi_allreduce(tmpDrv2, dtCDP , 1, mpi_real8, mpi_max, &
                   mpi_comm_world, ierr)

Readjust two Δt’s so that one becomes an integer factor of the other.
if(dtSUmb >= dtCDP) then
  nIterSUmb = 1
  nIterCDP = int(dtSUmb/dtCDP)
dtSUmb = dtCDP*real(nIterCDP)
else
  nIterCDP = 1
  nIterSUmb = int(dtCDP/dtSUmb)
dtCDP = dtSUmb*real(nIterSUmb)
endif

Reset Δt to the adjusted value via sumb_setParam:
if(groupName == 'SUMB') then
call sumb_setParam(dtSUmb, 'unsteady time step (in sec)')
else if(groupName == 'CDP') then
call cdp_setParam(dtCDP, 'DT')
endif

else if((.not. adjustDt) .and. n == 1) then
  nIterSUmb = 1
  nIterCDP = 1
endif

! Run actual iterations...
if(groupName == 'SUMB') then
call sumb_runIteration(nIterSUmb)
else if(groupName == 'CDP') then
call cdp_runIteration(nIterCDP)
endif

if(groupName == 'SUMB') then
  if(mod(n,nWriteStepsVolSUmb) == 0) &
     call sumb_writeVolumeSolutionFile
  if(mod(n,nWriteStepsSurSUmb) == 0) &
     call sumb_writeSurfaceSolutionFile
endif

if((((.not. movGridSUmb) .and. (.not. movGridCDP)) &
     or (n == nSteps)) cycle iter_loop
endif

! Setup the point stuffs where data will be requested by SUmb...
call sumb_getPointSize(nPoints, 'SUmbRhouvwInterfaceFamily')
allocate(xyz(3,nPoints))
call sumb_getPointGeom(xyz, 'SUmbRhouvwInterfaceFamily')
call chimps_setPointGeom(xyz, nPoints, 'SUMB_POINTS')
deallocate(xyz)

! Setup the SUmb mesh stuffs...
call sumb_getMeshSize(nNodes, nTetra, nPyra, nPrism, nHexa, &
                      'SUmbAirfoil')
allocate(syms3(3,nPoints))
call chimps_setPointGeom(syms, nPoints, 'SUmb_POINTS')
deallocate(syms)

else if(movGridCDP .and. groupName == 'CDP') then
  ! Setup the point stuffs where data will be requested by CDP...
call cdp_getPointSize(nPoints, 'cdp_interface')
allocate(syms3(3,nPoints))
call chimps_setPointGeom(syms, 'cdp_interface')
deallocate(syms)

  ! Set up the CDP mesh stuffs...
call cdp_getMeshSize(nNodes, nTetra, nPyra, nPrism, nHexa, &
                     'CDP_AIRFOIL')
allocate(syms3(3,nNodes), tetsConn(4,nTetra), pyraConn(5,nPyra), &
          prismConn(6,nPrism), hexaConn(8,nHexa))
call cdp_getMeshGeom(syms, tetsConn, pyraConn, prismConn, &
                     hexaConn, 'CDP_AIRFOIL')
call chimps_setMeshGeom(syms, tetsConn, pyraConn, nTetra, &
                         pyraConn, nPyra, prismConn, nPrism, &
                         hexaConn, nHexa, 'CDP_MESH')
deallocate(syms, tetsConn, pyraConn, prismConn, hexaConn)
endif
call cdp_getMeshSize(nNodes, nTetra, nPyra, nPrism, nHexa, &
  'default-interior')
allocate(xyz(3,nNodes), tetraConn(4,nTetra), pyraConn(5,nPyra), &
  prismConn(6,nPrism), hexaConn(8,nHexa))
call cdp_getMeshGeom(xyz, tetraConn, pyraConn, prismConn, &
  hexaConn, 'default-interior')
call chimps_setMeshGeom(xyz, nNodes, tetraConn, nTetra, &
  pyraConn, nPyra, prismConn, nPrism, &
  hexaConn, nHexa, 'CDP_MESH')
ddeallocate(xyz, tetraConn, pyraConn, prismConn, hexaConn)
endif
enddo iter_loop

! Finalize each solver...
if(groupName == 'SUMB') then
  call sumb_finalize()
else if(groupName == 'CDP') then
  call cdp_finalize()
endif

! Finalize CHIMPS...
call chimps_finalize()

! Terminate the MPI session...
call mpi_finalize(ierr)

! ********** End of the program **********
end program coupled_pitching_xf