一种开源多物理场耦合软件 EasyFSI

张兵[1,*]，周帆[1]，员海玮[2]

1. 合肥工业大学 机械工程学院，安徽省 合肥市 23009

2. 南京航空航天大学 航空学院，江苏 南京 210016

中图分类号：V211.3　　文献标识码：A

摘要：本文针对气动弹性等多物理场耦合计算的关键问题开展研究，对进程通信技术、模型耦合接口、物理量插值方法进行设计。实现了基于 Socket 的不同求解器进程通信，以及适用于 MPI 并行的同一求解器内部进程的通信方法；实现了多种耦合界面物理量插值方法；采用 C++ 语言开发了一种开源耦合分析软件 EasyFSI，并提供了 C/C++、Python 和 MATLAB 调用接口。算例结果表明该软件具有良好的插值精度和计算效率。

关键字：多物理场耦合, 开源, 气动弹性

中图分类号

基于 CFD/CSD 耦合的气动弹性、热气动弹性问题的求解需要将多个求解器耦合进行。开展此类工作，不仅需要准确实现耦合边界或计算域的自由度、载荷插值，还需要高效的进程通信技术，以及灵活的耦合迭代策略。随着计算规模的增加，各求解器往往需要采用 OpenMP、MPI 等并行技术加速，使其通信方法和耦合策略实现变得较为复杂。开发具有良好性能和精度，并适用于复杂多物理场耦合的计算软件具有重要的应用价值。

目前不少商业软件已具备多物理场耦合功能，如 MDICE[1]、MpCCI[2]、ANSYS Workbench[3] 以及 SIMULIA co-simulation engine[4][5]等，但此类软件往往存在闭源性、耦合策略不易调整、接口不开放等问题，难以适用于日益复杂的多物理场耦合计算要求。在开源社区中也涌现处了一批优秀代码，如 preCICE[6]，该开源软件为慕尼黑工业大学和斯图加特大学共同开发的面向多物理场耦合分析的软件，具备灵活的耦合计算策略和集成能力。

本文在作者已开发的基于 RPC 的多物理场耦合软件[7]基础上，改进了通信技术、并行处理能和二进制接口（ABI），并增加了 Python 和 MATLAB 接口，软件源代码托管在 github 上[8]。

1 进程通信方法

如图 1 所示为目前常见的多物理场耦合计算多软件交互示意图。每个求解器内部可能采用 OpenMP 多线程或 MPI 多进程并行，不同求解器之间通过进程通信技术实现模型数据的传递和同步。其核心功能依赖于

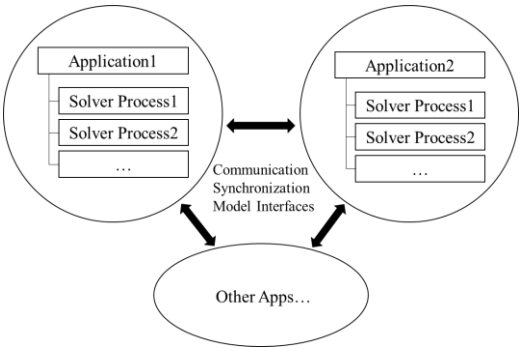


图 1 多物理场耦合软件交互

进程间通信（IPC）的底层技术主要包括文件（File I/O）、管道（Pip）、共享内存（Shared Memory，SHM）、套接字（Socket）、远程直接内存访问（RDMA）等；同时还可以使用对上述方法进行封装的远程过程调用（RPC）技术，如 Microsoft RPC[9]、gRPC[10]等。这些技术的优缺点如表 1 所示：

表 1 进程间通信技术对比

序号	名称	优点	缺点
1	File I/O	编程简单	受限于磁盘性能，效率往往较低*1

国家自然科学基金：12172116

2	Pip	编程简单	仅限于单个操作系统内的进程通信
3	SHM	通信效率高	编程复杂，仅限于单个操作系统内的进程通信
4	Socket	通信效率较高，跨操作系统通信	编程较复杂，跨系统通信时需要网络设施
5	RDMA	通信效率高	需专用硬件和软件设施
6	RPC	编程较简单	需要服务器-客户端模型，影响效率

注：1）可通过内存虚拟盘来提高性能

本文对进程通信功能进行抽象，定义为 Communicator 基类，如图 2 为其 UML 图，该对象提供 send 和 recv 两个抽象方法实现数据的发送和接收。

Communicator
+send(data,count,type,dest,tag) Pure Virtual
+recv(data,count,type,src,tag) Pure Virtual

图 2　Communicator 对象

所有基于底层 IPC 技术的通信方法均可通过从该对象派生来实现。目前本文实现了基于 Socket 的 Communicator 对象 SocketCommunicator，未来计划实现基于共享内存的 ShmCommunicator。基于 Socket 的通信过程如图 3 所示：

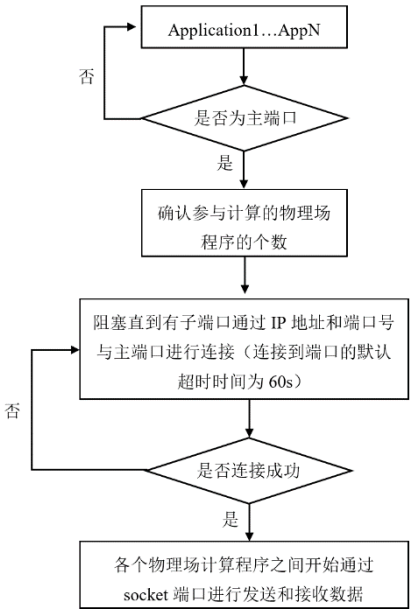


图 3 基于 Socket 技术的数据传递示意图

注意上述通信对象通常用于不同求解器间的数据通信，而对于同一求解器的不同进程或线程间的通信则同样可通过从 Communicator 基类派生定义。目前本文已实现了适用于 MPI 的 MPICommunicator，以及适用于 ANSYS Fluent 软件的 FluentCommunicator。

2. 模型接口及物理量插值方法

多物理场耦合计算需要传递的模型数据与特定问题相关，例如对于气动弹性问题，需要传递计算域耦合边界的位移、温度、力、热流等。本文对需要传递的模型信息统一定义为 ModelInterface 基类，其 UML 图如图 4 所示，通过纯虚函数 nnode、nelem 获取接口的节点和单元个数，通过 field 相关函数注册和获取场变量数据。

ModelInterface
+nnode() Pure Virtual
+nelem() Pure Virtual
+register_field(info) Pure Virtual
+nfield()
+get_field(field_name)

图 4　ModelInterface 对象

气动弹性计算所需的流固耦合边界信息可从 ModelInterface 派生，本文已实现了表示计算域网格耦合边界的 Boundary 对象，以及表示计算域的子区域的 CoupledRegion 对象。其中 Boundary 对象会在主进程中组装为单个 DistributedBoundary 对象用于进行物理量插值计算。

本文将耦合接口上的物理量插值算法定义为 Interpolator 对象，其 UML 图如图 5 所示：

Interpolator
+add_source_int(model_interface)
+add_target_int(model_interface)
+interp_dofs_s2t(ndof, dof_s, dof_t)
+interp_loads_t2s(nload, load_t, load_s)
+compute_coeff(method)

图 5　Interpolator 对象

本文将耦合物理量分为自由度（DOF，如位移、温度）和载荷（Load，如力和热流）两类，其中自由度只能从源接口（Source）

国家自然科学基金：12172116

插值到目标接口（Target），载荷则相反。通过插值器对象的 add_xxx 方法定义耦合的模型接口，通过 compute_coeff 方法计算插值系数，最后通过 interp_xxx 方法完成耦合接口的物理量插值。

物理量的插值方法会直接影响耦合计算精度。本文实现了如下四类插值方法，这些方法可在调用 compute_coeff 函数时通过参数指定：

1）全局样条插值方法（Global XPS）

2）局部样条插值方法（Local XPS），适用于无单元信息的点云接口；

3）等参逆变换方法（Projection），适用于含单元信息的网格边界或计算域接口；

4）几何映射方法（Mapping），适用于精确保持通量守恒的边界载荷插值。

3. 求解器对象与耦合策略

本文对求解器对象进行抽象，定义为 Application 类，其 UML 图如图 6 所示：

Application
+Application(name,intra_comm,root)
+register_field(name,ncomp,loc,io,units)
+set_field_function(getter,setter)
+add_interface()
+start_coupling(inter_comm)
+exchange_solution(time,obj)
+stop_coupling()

图 6 Application 对象

该对象通过构造函数中的 intra_comm 参数设置同一求解器内部不同进程间的 Communicator 对象（如 MPICommunicator、FluentCommunicator），通过 add_interface 添加耦合接口（如 Boundary、CoupledRegion），通过 exchange_solution 来完成不同求解器间的数据插值和传递。求解器进程间的数据通信如图 5 所示：

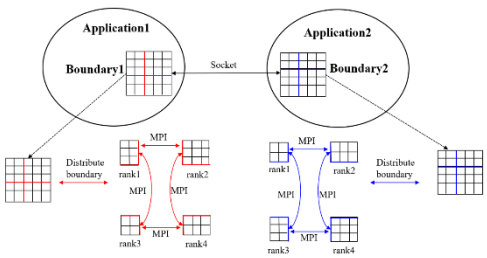


图 7 求解器进程间的数据通信

为了适用于复杂耦合策略，EasyFSI 提供了 C/C++、Fortran、python 和 MATLAB 接口，可以在应用程序内部根据策略需要以任意组合方式调用 exchange_solution 函数实现。

本项目编译目标为静态和动态链接库，可方便的在 C/C++、Fortran 求解器中通过静态或动态链接使用 EasyFSI 功能。下图 7 给出了在 C/C++求解器中使用 EasyFSI 的演示（demo.cpp），其主要步骤如下：

1）定义各对象：3~6 行；

2）定义边界物理量读写函数：12~64 行；

3）初始化各个对象：70~106 行；

4）求解过程数据传递：118 行；

5）后处理对象清理：130~134 行

```cpp
#include "EasyFsi.h"

static Application*      app        = nullptr;
static Communicator*     inter_comm = nullptr;
static MPICommunicator*  intra_comm = nullptr;
static Boundary*         bd0        = nullptr;

//-----------------------------------------------------------
// define helper functions for field reading and writing
//-----------------------------------------------------------

//! @brief function used to reading outgoing fields invoked by Application
void get_boundary_field(const Application* app, const Boundary* bd, const char* name, int
ncomp, FieldLocation loc, double* data, void* user_data)
{

//! @brief function used to writing incoming fields invoked by Application
void set_boundary_field(const Application* app, const Boundary* bd, const char* name, int
ncomp, FieldLocation loc, const double* data, void* user_data)
{

//-----------------------------------------------------------
// preprocessing
//-----------------------------------------------------------

void init()
{
    // create communicator used to transfer data between different application.
    inter_comm = cm_socket_new(true,2,"127.0.0.1",1234);

    // create MPI communicator used to transfer data between process of this application
    //    arg0 : The MPI communicator
    //    arg1 : Rank of this process in mpi_comm
    //    arg2 : Total process number of mpi_comm
    intra_comm = cm_mpi_new(mpi_comm, rank, size);

    // create application
    app = app_new("cfd",intra_comm,0);

    // define coupled boundary
    bd0 = app_add_boundary(app);
    // 1) create boundary manually:
    // bd_add_node(bd0,x1,y1,z1,id1);
    // ...
    // bd_add_face(bd,ft1,nn1,nodes1);
    // ...
    //
    // 2) create boundary from Gmsh file:
    // bd_read_gmsh(bd, "???.msh");

    bd_compute_metrics(bd0, 5.0);

    // define coupled fields:
    app_register_field(app,"displacement",3,NodeCentered,OutgoingDofs ,"m");
    app_register_field(app,"force"       ,3,NodeCentered,IncomingLoads,"N");

    // set field reading/writing functions
    app_set_field_func(app,get_boundary_field,set_boundary_field);

    // start coupling: get solver infomation
    app_start_coupling(app,inter_comm);
}

//-----------------------------------------------------------
// solving
//-----------------------------------------------------------

void solve()
{
    // solve this physics one step
    // ...

    // interpolate and exchange results between applications
    app_exchange_solu(app,time,nullptr);

    // other operations
    // ...
}

//-----------------------------------------------------------
// postprocessing
//-----------------------------------------------------------

void post()
{
    app_stop_coupling(app);
    app_delete(&app);
    bd_delete(&bd0);
    cm_delete(&intra_comm);
    cm_delete(&inter_comm);

    // other operations
}
```

国家自然科学基金：12172116

图 7　C/C++求解器中的耦合演示

本项目使用 pybind11 进行 Python 语言接口的创建，如图 8 为 Python 中的使用演示（demo.py），其关键步骤与 C/C++演示一致。关键步骤包括：边界物理量读写函数定义、前后处理、求解过程物理量的插值及传递。

```python
import EasyFsi
from EasyFsi import*

#-----------------------------------------------------------
# define helper functions for field reading and writing
#-----------------------------------------------------------

# function used to read outgoing field of boundary.
#    app: application object
#    bd: boundary object
#    fieldname: name of the field
#    location: location of the field, see FieldLocation
#    values: field data, type=MatView object
#    user_obj: object passed from app.exchange_solution
def get_bound_field(app,bd,fieldname,ncomp,location,values,user_obj):
    # TODO: update values
    for i in range(bd.nnode):
        values[i,0]=???; # update component-0
        values[i,1]=???; # update component-1
        #...

# function used to write incoming field of boundary.
#    app: application object
#    bd: boundary object
#    fieldname: name of the field
#    location: location of the field, see FieldLocation
#    values: field data, type=MatView object
#    user_obj: object passed from app.exchange_solution
def set_bound_field(app,bd,fieldname,ncomp,location,values,user_obj):
    # TODO: update values
    for i in range(bd.nnode):
        # ... = values[i,0]; # update component-0
        # ... = values[i,1]; # update component-1
        #...

#-----------------------------------------------------------
# preprocessing
#-----------------------------------------------------------

# define application
app = Application("PythonSolver");

# define boundary
bd0 = app.add_coupled_boundary()
bd0.name = "bd0"
# create boundary manually:
#bd0.reserve(200,100,400)
#bd0.add_node(x,y,z,unique_id) # define node
# ...
#bd0.add_face(type,nodes)  # define face
# ...
# create boundary from file:
#bd0.load("???.msh")  # read Gmsh file
#bd0.compute_metics(5.0)

# define field
#    arg0  The name of this field
#    arg1  The component number of this field, >=1
#    arg2  Location of field, NodeCentered or FaceCentered
#    arg3  Input/Output type, see FieldIO
#    arg4  Units of this field
app.register_field("displacement",3,FieldLocation.NodeCentered,FieldIO.OutgoingDofs,"m")
app.register_field("force",3,FieldLocation.NodeCentered,FieldIO.IncomingLoads,"N")
app.set_field_function(get_bound_field,set_bound_field)

# define communicator between applications
#    arg0  True/False, this application is master?
#    arg1  Number of applications for this coupling problem, >=2
#    arg2  IP address of machine that master application is running.
#    arg3  Port number
#    arg4  Timeout value in second.
inter_comm = SocketCommunicator(False,2,"127.0.0.1",1234,60)

# start coupling
app.start_coupling(inter_comm)

#-----------------------------------------------------------
# solving
#-----------------------------------------------------------

# solving
dt = 0.001  # timestep size
nt = 1000   # timestep number
for it in range(nt)
    # TODO: solve one step
    #...

    # exchange solution
    app.exchange_solution(dt*(it+1), None)

    # other post operations
    #  ...

# stop coupling when finished
app.stop_coupling()

#-----------------------------------------------------------
# postprocessing
#-----------------------------------------------------------

# save results
app.save_tecplot("pysolver_res.plt")
```

图 8　python 求解器中的耦合演示

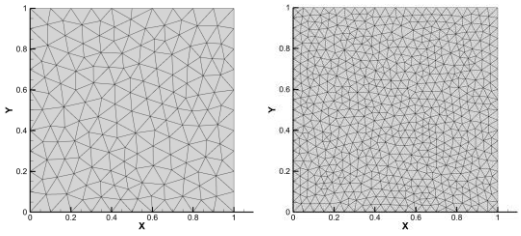
3 算例测试与结果分析

1）插值方法验证

采用单位正方形区域作为耦合边界，验证插值方法的精度，场变量假设为采用如下形式：

$$f(x,y) = sin\frac{x}{\pi}cos\frac{x}{\pi}$$

插值边界网格分别为单元尺寸 0.1 和 0.05 的非结构，如图 9 所示；测试插值方法的 Python 脚本如图 10 所示。



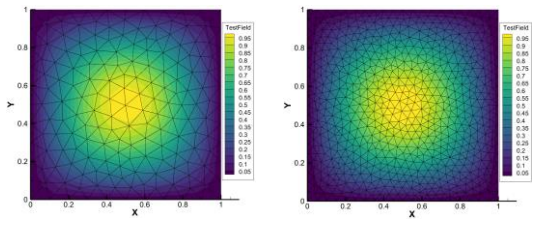
(a) 源边界　　(b) 目标边界

图 9　插值边界网格

```python
import math
import EasyFsi
from EasyFsi import*

# create FieldInfo
disp_s = FieldInfo("displacement","m",3,FieldLocation.NodeCentered,FieldIO.OutgoingDofs)
disp_t = FieldInfo("displacement","m",3,FieldLocation.NodeCentered,FieldIO.IncomingDofs)
force_s = FieldInfo("force","N",3,FieldLocation.NodeCentered,FieldIO.IncomingLoads)
force_t = FieldInfo("force","N",3,FieldLocation.NodeCentered,FieldIO.OutgoingLoads)

# create boundary
bound_s = Boundary()
bound_t = Boundary()

bound_s.load("fe.msh")
bound_t.load("fv.msh")
bound_s.register_field(disp_s)
bound_s.register_field(force_s)
bound_t.register_field(disp_t)
bound_t.register_field(force_t)

# create interpolator
interp = Interpolator()
interp.add_source_boundary(bound_s)
interp.add_target_boundary(bound_t)
interp.compute_interp_coeff(InterpolationMethod.LocalXPS,20)
interp.save_coefficients("coeff.txt")

# setup field value
disp = bound_s.get_field("displacement")
for i in range(0,bound_s.nnode):
    coord=bound_s.node_coords(i)
    disp[1,2]=math.sin(coord.x*math.pi)*math.sin(coord.y*math.pi)

# do interpolating
interp.interp_dofs_s2t("displacement")
interp.interp_load_t2s("force")

# save results
bound_s.save("fe.plt")
bound_t.save("fv.plt")
```

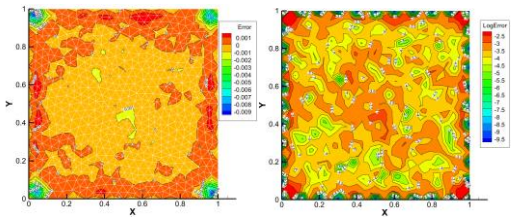
图 10　插值测试脚本

插值结果及误差如图 11 所示，可以看出插值结果与源场变量分布具有相似的规律，插值误差的绝大部分区域的绝对误差量级均在 10^{-3} 以下，表明本文插值方法具有良好的计算精度。



(a) 源场　　(b) 插值目标场

国家自然科学基金：12172116



(c) 绝对误差　　(d) 对数误差

图 11　插值结果及误差分布

2）气动弹性算例

本算例模型为在 NASA Langley 9×18 in 超声速风洞进行的超声速颤振实验[11]，模型编号为 Model-2B，几何外形为根部固支的切尖三角翼，材质为铝合金 2024-T3，来流马赫数为 3.0。计算有限元和 CFD 网格如图 12、13 所示。

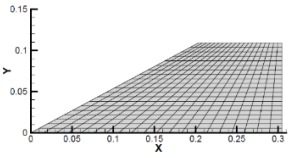
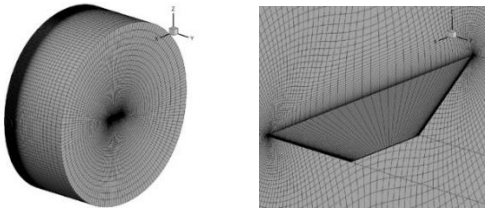
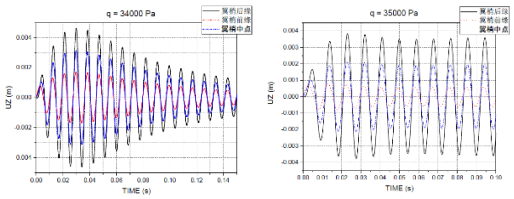


图 12 结构有限元网格

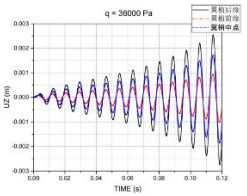


(a)整体视图　　(b)局部视图

图 13 CFD 网格



(a)动压 34000Pa　　(b)动压 35000Pa



(c)动压 36000Pa

图 14 不同来流动压下的位移响应

如图 14 所示为收敛、临界发散、发散三种状态的监视结点 z 方向位移响应时间历程，来流动压为 35000Pa 时为接近颤振临界状态，这与试验颤振动压 33995Pa 非常接近，表明分析本文方法具备较高的计算精度。

表 2　计算时间统计

求解器	时间/s	参数说明
EasyCFD	682	82 万网格，32 核 MPI 并行，9 万迭代步
MSP	1.6	5 阶模态叠加，单核，3000 步
EasyCouple	1.5	约 1.2 万次 RPC 调用
EasyFSI	0.6	约 0.6 万次 Socket 调用

计算硬件配置为 2 个 2×Intel Xeon E2650 8 核计算结点，通过 56GB/s Infiniband 网络链接，其中 CFD 采用全部的 2 个节点的 32 个 CPU 核心，模态叠加法程序在一个结点上串行运行。在 0.1s 物理时间下，流固耦合各子系统消耗的时间如表 1 所示。可以看出，与基于 RPC 方法的 EasyCouple 相比，EasyFSI 的数据传递时间降低了一半以上，仅占总分析时间的 0.09%，说明本文方法具备优良的计算效率。

结论

本文开发了一个适用于多物理场耦合的开源软件 EasyFSI，适用于分布式并行的多个求解器之间的高效耦合分析，具有良好的耦合接口、插值精度和计算效率。可应用于气动弹性、热气动弹性、共轭传热等流固耦合问题的仿真分析。

参考文献

[1] John, M.S., Jr. V. P., Gerry, M. K., et al. Application of a Multi-Disciplinary Computing Environment (MDICE) for Loosely Coupled Fluid-Structural Analysis[R]. AIAA-98-4865.

[2] http://www.mpcci.de/mpcci-software.html, 2015.06.

[3] http://www.ansys.com/Products/Workflow+Technology/ANSYS+Workbench+Platform, 2015.06.

[4] Blades E, Miskovish S, Luke E, et al. A Multiphysics Simulation Capability using

国家自然科学基金：12172116

the SIMULIA Co-Simulation Engine[R]. AIAA-2011-3397.

[5] Scheibe C, Cemerow A, et al. A Novel Co-Simulation Concept using Interprocess Communication in Shared Memory, 2019 IEEE Power & Energy Society General Meeting (PESGM), Atlanta, GA, USA, 2019:1-5

[6] https://github.com/precice/precice, 2023

[7] 张兵，王华毕，基于RPC技术的多场耦合分析软件研究[C]，第十四届全国空气弹性学术会议，中国，西安，2015-08-23至2015-08-25

[8] https://github.com/ZHBHFUT/EasyFsi, 2023

[9] https://learn.microsoft.com/en-us/windows/win32/rpc/rpc-start-page, 2023

[10] https://grpc.io/, 2023

[11] Perry W H, Gilbert M L. Experimental and calculated results of a flutter investigation of some very low aspect-ratio flat-plate surfaces at Mach number from 0.62 to 3.00[R]. NASA-TN-D-2038, 1963.

国家自然科学基金：12172116