THE UNIVERSITY of EDINBURGH

Edinburgh Research Explorer

OpenFPCI: A parallel fluid–structure interaction framework

OPEN ACCESS
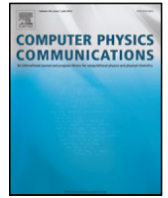
OpenFPCI: A parallel fluid–structure interaction framework☆

Sam Hewitt [a], Lee Margetts [a,*], Alistair Revell [a], Pankaj Pankaj [b],
Francesc Levrero-Florencio [b,c]

[a] *School of Mechanical, Aerospace and Civil Engineering, The University of Manchester, Manchester, M13 9PL, United Kingdom*
[b] *Institute for Bioengineering, School of Engineering, The University of Edinburgh, Edinburgh, EH9 3DW, United Kingdom*
[c] *Department of Computer Science, University of Oxford, Oxford, OX1 3QD, United Kingdom*

ABSTRACT

This paper presents OpenFPCI, a framework for coupling the C++ toolbox OpenFOAM-Extend, a computational fluid dynamics package, with the general purpose finite element package ParaFEM, written in Fortran and used to solve structural mechanics problems. The coupling of these two open source and scalable toolboxes, facilitates the use of high performance computing resources for the solution of fluid–structure interaction problems. The framework uses a master–slave approach, with OpenFOAM-Extend acting as the master and calling OpenFPCI plugins. The plugins are composed of a series of subroutines used to initialise and solve a specific engineering problem and make use of ParaFEM's highly parallel implementation. The plugins are wrapped by C constructs such that OpenFOAM-Extend can call these Fortran subroutines consistently and when the solution from ParaFEM is required. Each plugin solves a different solid mechanics problem, with the current features including the deformation of a linear-elastic structure undergoing small strain and the deformation of a St. Venant–Kirchhoff material. Throughout this paper the focus will lie on the large strain plugin, considering the implementation and its validation for a benchmark problem, along with assessment of parallel capabilities, which are shown to scale to three thousand cores. This paper will be of interest to OpenFOAM and ParaFEM practitioners looking to utilise multiphysics simulations for their research, along with researchers looking to integrate fluid–structure interaction into their studies.

Program summary

Program Title: OpenFPCI
Program Files doi: http://dx.doi.org/10.17632/ntprzxk477.1
Licensing provisions: BSD 2-Clause
Programming language: Fortran, C and C++
External libraries: OpenFOAM and ParaFEM
Supplementary material: Example test cases are available within the OpenFPCI repository.
Nature of problem: OpenFPCI was developed to solve computationally expensive fluid structure interaction problems by running on high performance computing systems. The framework was designed to enable the coupling of advanced ParaFEM capabilities to OpenFOAM-Extend.
Solution method: OpenFOAM-Extend uses the classic Arbitrary Lagrangian–Eulerian formulation of the Navier–Stokes equations to deal with moving boundaries. The moving boundary is defined by using an OpenFPCI plugin, using ParaFEM's libraries, to solve the deformation of the adjoining structure.

1. Introduction

Fluid–structure interaction (FSI) simulates two physical systems, the fluid and structure and the effect they have on each other. In many cases the interdependent effect each domain has on the other cannot be ignored, fluid forces exerted on a structure can result in significant structural deformation, that modifies the fluids motion. FSI methods are used in a variety of research fields, including biomedicine [1–3] and aerodynamics [4–6], with researches highlighting the improved predictions achieved through multiphysics simulations. In biomedicine Scotti and Finol [1] studied the impact of using FSI methods on the rupture of abdominal aortic aneurysms. They compared the use of a rigid wall against a deforming one, on patient specific geometries, summarising that simulations with flexible walls offer a more

accurate predictor of potential aneurysm rupture. Researchers have considered the effects of using FSI methods in wind turbine aerodynamics. Hsu and Bazilevs [4] and Korobenko et al. [5], both highlighted that FSI was required to accurately predict the cyclic loading on the turbine blades, ultimately providing improved approximations of turbine blade fatigue life. The use of FSI in modelling and simulation is imperative to improve the level of detail and physical realism required in many fields today [7].

Two approaches exist to solving FSI problems, monolithic methods [8] and partitioned methods [9]. Monolithic approaches combine the governing equations for both physical systems and formulate a single set of equations, that is solved each time step. In comparison, partitioned methods solve the governing equations for each sub-domain independently with the interaction between them acting as a boundary condition to the other domain. Monolithic methods are widely accepted to offer improved stability and accuracy [10], however the application of these methods is problem dependent, the formulated single set of equations that represents the combined impact of the fluid and solid, needs to be modified for each problem. Furthermore, a range of well established, single physics packages, that are well tailored to the scope of their problem, already exist. These single physics packages, that have over twenty years of development, can be easily used within partitioned approaches, that are inherently modular. Coupling two single physics packages through an interface, makes use of the extensive research that has gone into each of them and aggregates the robust and efficient capabilities to solve single physics problems, into a robust and efficient package to solve multiphysics problems.

Commercial vendors have begun to offer FSI simulation capabilities within their packages, ADINA [11], ANSYS [12] and COMSOL Multiphysics [13] to name a few. However the black-box nature of these applications has led researchers to develop their own open source alternatives. The alternatives can be categorised into direct coupling approaches [14] and approaches using general purpose coupling environments [15].

Direct approaches, using two independent packages, often use a master–slave scheme. One of the packages is chosen as the master and makes calls to the external libraries from the other package. A number of authors have used OpenFOAM as a CFD package with which to couple a structural package. Lorentzon [14] interfaced DEAL.II, a C++ finite element package with OpenFOAM and Cesur et al. [16] coupled OOFEM, another C++ finite element package. OpenFOAM-Extend has its own staggered FSI capabilities developed by Tukovic et al. [17], solving both the fluid and solid domains using the finite volume method.

General purpose coupling environments for partitioned multiphysics problems are an alternative to these direct approaches. The environments deal with the data mapping, coupling strategy and communication between the solvers. CoMA [18] has been used by Breuer et al. [19] to couple their in house CFD code FASTEST-3D with their in house finite element solver, Carat++, designed to model shell and membrane behaviour. Gallinger [18] used CoMA, to interface Carat++ with OpenFOAM. PreCICE [20] is another open source alternative that uses a peer-to-peer approach in comparison to the server based approach used by CoMA and MpCCI [21], a commercial application focusing on the interfacing of commercial packages with pre-existing adapters. Server based approaches use a centralised server to manage the work flow between the packages, and can be a bottleneck as the size of the interface work grows. Peer-to-peer approaches, split the work typically preformed by the centralised server across the peers, so that the interface work is completed predominantly in parallel.

In this paper we present a new package, OpenFPCI, Open source Foam to ParaFEM Coupling Interface, which couples the open source, highly parallel finite element toolbox ParaFEM [22] with a recently developed FSI library [17] available as part of OpenFOAM-Extend, an extension to the original OpenFOAM developed by Weller et al. [23].

ParaFEM, written in Fortran, is a collection of libraries and highly parallel mini-apps [24]. Each mini-app solves a specific engineering problem, with a range of capabilities including nonlinear material behaviour (plasticity) [25], geometric nonlinearity [26], multiscale fracture [27], thermomechanical analysis [28] and stochastic Monte Carlo Simulation [29]. A lean procedural programming style is used within each mini-app which results in each program achieving good scalability using up to many tens of thousands of cores [22].

In comparison to OpenFOAM, OpenFOAM-Extend has placed more focus on integrating user-developed tools and applications. The toolbox developed in C++ is highly modular and flexible with capabilities to solve a range of complex flow phenomenon, along with its extensive pre-processing and post-processing utilities.

OpenFPCI provides a differing set of capabilities over the commercial packages and the open source applications described above. One of the key objectives when developing this interface was to solve FSI problems utilising High Performance Computing (HPC) facilities. Commercial packages may offer a larger range of capabilities, however they typically do not scale well on HPC platforms. Furthermore, the licencing costs place a limitation on the use of HPC facilities, as hardware and software grow towards exascale computing the cost and feasibility of purchasing huge numbers of licences for use across millions of cores is a major constraint. In this work, a direct coupling strategy was preferred to the use of one of the general purpose coupling environments, as it provides the developer with greater control of the data passing between the two packages. A further goal developing OpenFPCI was to isolate researchers focused on structural mechanics from the complexities associated with code coupling and create a quick and easy development platform to develop/integrate advanced structural models within an FSI framework.

This paper provides a comprehensive description of the framework and current solid mechanics capabilities developed within OpenFPCI. Fig. 1 provides a summary of the framework that will be described in the paper, with each OpenFPCI plugin solving a particular solid mechanics problem. The motivation for opening-up and disseminating this research is not only to encourage the use of its current capabilities but enable researchers to further extend the capability by implementing state of the art techniques for computational mechanics using ParaFEM. The detailed description of the OpenFPCI framework is provided from the point of view of an OpenFPCI plugin, used to solve the deformation of a geometrically nonlinear St. Venant–Kirchhoff material undergoing large strain.

The paper is organised as follows, Section 2 details the underlying governing equations of FSI problems, followed by a description of the implementation of OpenFPCI, in Section 3. Sections 4 and 5 detail the validation and performance of the OpenFPCI plugin respectively, before an example of extending the framework is provided in Appendix A. Finally the installation instructions and an example test case are provided in Appendices B and C.

2. Methodology

Modern software design approaches often focus on the reuse of existing software that is already highly optimised for its purpose. Partitioned approaches conform to these ideologies, allowing the reuse of existing packages. In the context of the modelling and simulation of FSI problems, the fluid and solid domains are formulated and discretised using techniques common in their
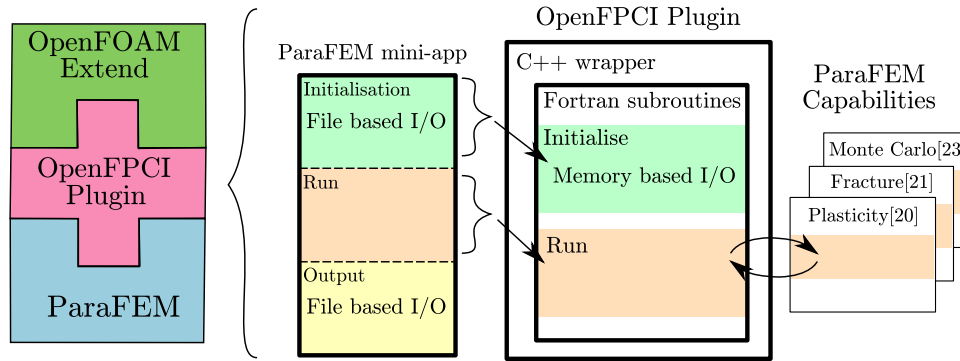
Fig. 1. Summary of an OpenFPCI plugin, highlighting the decomposition of a ParaFEM mini-app into two subroutines, initialise and run. In the development of a new OpenFPCI plugin only the run routine needs to be swapped and an example of ParaFEM's capabilities is shown for possible mini-app developments.

respective fields. The numerical methodology is therefore split into three areas; the solid mechanics problem, the fluid mechanics problem and the interface between them.

2.1. Notation

Tensor notation is used throughout this manuscript, with indicial notation within brackets being used in this subsection to clarify certain tensorial operations, or in specific sections where further clarification might be required.

As a general rule, scalars are denoted with Greek or Latin italic characters (e.g. α or a, respectively); vectors, or first-order tensors, are denoted by Latin bold lower-case characters (e.g. $\mathbf{a}$); second-order tensors are denoted with Greek or Latin bold upper-case characters (e.g. Ω or $\mathbf{A}$, respectively).

Tensorial operations are denoted as follows. The gradient of a scalar field is the vector field $\nabla a \left((\nabla a)_i = \frac{\partial a}{\partial X_i} \right)$, where X_i are the material (undeformed) coordinates of the system. The divergence of a vector field is the scalar field $\nabla \cdot \mathbf{a} \left(\nabla \cdot a = \frac{\partial a_i}{\partial X_i} \right)$. The gradient of a vector field is a second-order tensor field $\nabla \mathbf{a} \left((\nabla a)_{ij} = \frac{\partial a_i}{\partial X_j} \right)$. The divergence of a second-order tensor field is the vector field $\nabla \cdot \mathbf{A} \left((\nabla \cdot A)_i = \frac{\partial A_{ij}}{\partial X_j} \right)$. The trace of a second-order tensor is denoted as $\mathrm{tr}(\mathbf{A})$ $a = (A_{11} + A_{22} + A_{33})$. The transpose of a second-order tensor is denoted as $\mathbf{A}^{\mathrm{T}}$ $\left(A_{ij}^{\mathrm{T}} = A_{ji} \right)$. The single contraction of two vectors is denoted as $\mathbf{a} \cdot \mathbf{b}$ $c = (a_i b_i)$. Single contraction of two second-order tensors is denoted as $\mathbf{A}\mathbf{B}$ $C_{ij} = (A_{ik} B_{kj})$.

2.2. Solid mechanics

Within continuum mechanics the dynamic equilibrium of a structure is described by the conservation of momentum, which is provided in Eq. (1), in the Lagrangian reference frame.

$$\rho_s \frac{\partial^2 \mathbf{u}_s}{\partial t^2} = \nabla \cdot (\mathbf{F}\mathbf{S}) + \rho_s \mathbf{b}_0, \tag{1}$$

where subscript s indicates the property of the solid domain, $\mathbf{u}_s$ the displacement vector, $\mathbf{F} = \mathbf{I} + \nabla \mathbf{u}_s$ the deformation gradient, ρ_s the density, $\mathbf{S}$ the second Piola–Kirchhoff stress tensor and $\mathbf{b}_0$ the body forces given in the reference configuration. The constitutive law for a St. Venant–Kirchhoff material is used with the Second Piola–Kirchhoff stress given by Eq. (2).

$$\mathbf{S}_s = \lambda \, \mathrm{tr}(\mathbf{E}_s) + 2\mu \mathbf{E}_s, \tag{2}$$

where λ and μ are Lamé coefficients, $\mathbf{E}_s = \frac{1}{2}(\mathbf{F}^{\mathrm{T}}\mathbf{F} - \mathbf{I})$ is the Green–Lagrange strain tensor and $\mathbf{I}$ is the second-order identity tensor.

2.2.1. Discretisation

The decoupled nature of partitioned approaches means independent time integration and solution methodologies can be utilised for each domain. ParaFEM's mini-apps incorporate a number of time marching schemes, however within the large strain plugin, the Newmark [30] method is implemented. The standard Newmark-Beta formulas, with constants β and γ, are shown in Eqs. (3) and (4).

$$\dot{\mathbf{u}}_{n+1} = \dot{\mathbf{u}}_n + \Delta t \left[(1 - \gamma)\ddot{\mathbf{u}}_n + \gamma \ddot{\mathbf{u}}_{n+1} \right], \tag{3}$$

$$\mathbf{u}_{n+1} = \mathbf{u}_n + \Delta t \dot{\mathbf{u}}_n + \Delta t^2 \left[\frac{1 - 2\beta}{2} \ddot{\mathbf{u}}_n + \beta \ddot{\mathbf{u}}_{n+1} \right], \tag{4}$$

where n and $n + 1$ represent the current and subsequent time step respectively, and $\mathbf{u}$ the displacement vector. Although direct procedures to solve linear equations are common within the finite element method, iterative procedures provide more efficient memory storage for larger problems [31] and element-by-element iterative strategies are easy to parallelise with good load balancing [32]. Here, the element-by-element preconditioned conjugate gradient (PCG) method is used.

2.3. Fluid mechanics

The physics within the fluid domain is often represented in its Eulerian form. However, with the deforming boundary at the solid, an Arbitrary Lagrangian–Eulerian (ALE) formulation is used for the conservation of mass and momentum, as shown in Eqs. (5) and (6) respectively.

$$\nabla \cdot \mathbf{v}_f = 0, \tag{5}$$

$$\rho_f \left[\frac{\partial \mathbf{v}_f}{\partial t} + (\mathbf{v}_f - \mathbf{v}_b) \cdot \nabla \mathbf{v}_f^{\mathrm{T}} \right] = -\nabla P + \nu \nabla^2 \mathbf{v}_f, \tag{6}$$

where $\mathbf{v}_f$ represents the velocity vector of the fluid, t the time, $\mathbf{v}_b$ the interface boundary velocity, ρ_f the fluid density, P the pressure and ν the fluid viscosity.

2.3.1. Mesh motion

With ALE methods boundary of the fluid mesh deforms with the structural deformation. In order to preserve the quality and validity of the entire mesh within the fluid domain, the internal cells are moved [33]. This is performed by solving the Laplacian of the cell velocity [34],

$$\nabla \cdot (\gamma \nabla \mathbf{v}_b) = \mathbf{0}, \tag{7}$$

where γ represents a diffusivity constant and $\mathbf{v}_b$ the mesh deformation velocity. A number of mesh motion options are present and have been explored by Jasak and Tukovic [35].

2.3.2. Discretisation

OpenFOAM-Extend's implementation of discretisation methods allows the interchange of time stepping schemes and solution procedures through case files. Time schemes include 1st order implicit Euler, 2nd order implicit backward and 2nd order implicit and bounded Crank–Nicolson.

Unsteady problems within OpenFOAM are solved using the Pressure Implicit with Splitting Operator (PISO) procedure [36], which solves the standard pressure–velocity coupling problem through, (i) a momentum predictor, (ii) a pressure solution and (iii) a momentum corrector.

2.4. Interface

The coupling between the two applications requires satisfying two interface conditions. First kinematic equilibrium, Eq. (8), which ensures the geometrical domains continually match throughout the solution process.

$$\mathbf{v}_f = \frac{\partial \mathbf{u}_s}{\partial t}. \tag{8}$$

Second, the transfer of forces between the domains must satisfy equilibrium. This is achieved through the expression for dynamic equilibrium shown in Eq. (9).

$$\boldsymbol{\sigma}_f \cdot \mathbf{n}_f = -\left(J^{-1}\mathbf{FSF}^{\mathrm{T}}\right) \cdot \mathbf{n}_s, \tag{9}$$

where $\boldsymbol{\sigma}_f$ represents the stress tensor of the fluid at the wall, and $\mathbf{n}_f$ and $\mathbf{n}_s$ are the unit normals at the fluid and solid sides of the interface respectively.

3. Software implementation

This section is summarised by three main subsections. Firstly Section 3.1 provides a summary of the external FSI library with which an OpenFPCI plugin can be used. Secondly Section 3.2 introduces OpenFPCI with a high level look at the interoperability of the Fortran and C source code, along with a general summary of the OpenFPCI files and file structure. Finally Section 3.3 describes in more detail the major subroutines and methods used within the interface.

3.1. FSI library

An FSI library, developed by Tukovic et al. [17] at the University of Zagreb, has been released. The library allows the integration of externally written solvers for fluid dynamics and structural mechanics problems. This is achieved through two abstract classes describing the required implementation of a solid solver, *solidSolver.C* and fluid solver, *fluidSolver.C*. A further class is implemented to couple the two solvers and deal with the interface between the two, *fluidSolidInterface.C*. When comparing such a library to FSI applications that use general coupling environments, such as those described in the introduction, the *solidsolver* and *fluidsolver* classes act as the fluid and solid solvers respectively with *fluidSolidInterface* class replacing the functionality and purpose of the general coupling environment. This includes the data mapping, interpolation and coupling schemes.

The general software implementation of the strongly coupled Gauss–Seidel FSI iteration scheme [37] is shown in Fig. 2. The figure shows the pseudo steps (blue) of the executable program and specifically highlights method calls that wrap OpenFPCI subroutines (green). Each of these method calls is described in the following section with how they integrate OpenFPCI subroutines within them.

Each of the *Update Fluid Mesh*, *Solve Fluid Gov. Eqns* and *Solve Solid Gov. Eqn* processes solves a set of governing equations. The *Update Fluid Mesh* is associated with Eq. (7), *Solve Fluid Gov. Eqns*
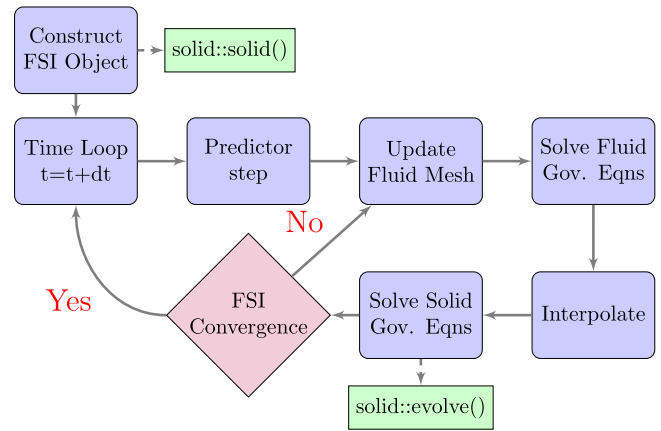

Fig. 2. General software implementation of the FSI algorithm, with the method calls to OpenFPCI routines highlighted in green. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Eqs. (5) and (6) and *Solve Solid Gov. Eqn* Eq. (1). Subsequently a description of the *Interpolate* and *FSI convergence blocks* is provided below.

3.1.1. Interpolation

OpenFOAM-Extends capabilities include interpolation techniques between two mesh surfaces. The interpolation between two non-conformal meshes takes place in two phases. The forces at the face centres on the surface of the fluid mesh are first interpolated to the surface face centres of the solid mesh. This is performed using the generalised grid interface developed by Beaudoin and Jasak [38], it uses a weighted interpolation between the two interface patches. The second step is to interpolate the face centre values to the nodes of each face. This is completed using a weighted method, making the interpolation process entirely conservative.

3.1.2. FSI convergence

The convergence of the strongly coupled iterative scheme ensures that the dynamic equilibrium between the fluid and solid solver is numerically accurate. Convergence is achieved once the L_2 norm of the residual, r_k, falls below its tolerance, ε_{fsi}. The residual is based on the difference between the displacements, at the interface, of the structure at two subsequent iterations. The L_2 norm of the displacement increment is normalised by the square root of the reference length and the number of nodes, at the interface, in the structure, Eq. (10) shows the calculation of the residual for a single time step.

$$\|\mathbf{r}_{k+1}\|_2 = \frac{\left\|\tilde{\mathbf{x}}_{k+1} - \mathbf{x}_{k+1}\right\|_2}{\sqrt{n \times L_{ref}}} \leq \varepsilon_{fsi}, \tag{10}$$

where $\mathbf{x}_{k+1}$ represents the displacement vector of the interface at the start of iteration $k + 1$, and $\tilde{\mathbf{x}}_{k+1}$ is the displacement of the interface predicted by the structural solver at iteration $k + 1$. L_{ref} and n are the problem reference length and the number of nodes in the interface vector respectively. The tolerance and reference length are user-tunable parameters that are altered on a case-by-case basis.

The convergence of this fixed point iteration scheme has been shown to improve through the use of relaxation [39]. A relaxation scheme is utilised such that at each iteration the mesh is not displaced the same distance as that calculated by the structural solver. Eq. (11) shows the relaxation relationship.

$$\mathbf{x}_{k+1} = \omega_k \tilde{\mathbf{x}}_k + (1 - \omega_k)\mathbf{x}_k. \tag{11}$$

Table 1

A summary and description of the OpenFPCI source files and their purposes.

Summary	Subroutine	Description
Derived class	femLargeStrain.*	Derived class for the large strain plugin, it wraps parafemnl.f90
	femSmallStrain.*	Derived class for small strain plugin, it wraps parafeml.f90
Include file	updateForce.H.	Included within the derived classes and contains the source code to interpolate forces from cell centres to nodal values using a weighted conservative method
Solvers	parafemnl.f90	The file contains two subroutines, runnl and initnl. An array of external forces is input to the runnl subroutine, that uses the Newmark method and Newton–Raphson iterations to step in time and solve the deformation of an elastic, geometrically non-linear structure. Displacement, velocity and acceleration arrays are output
	parafeml.f90	The file contains two subroutines, initl, runl. The runl subroutine accepts an array of external forces, and uses a linear interpolation in time to solve the deformation of a linear elastic structure, using the small strain assumption. Displacement, velocity and acceleration arrays are output
Utilities	parafemutils.f90	Utilities for interfacing between C++ and extensions to capabilities such as gravitational loading. It also contains debugging routines to print timings and variables to files and track the memory of a program
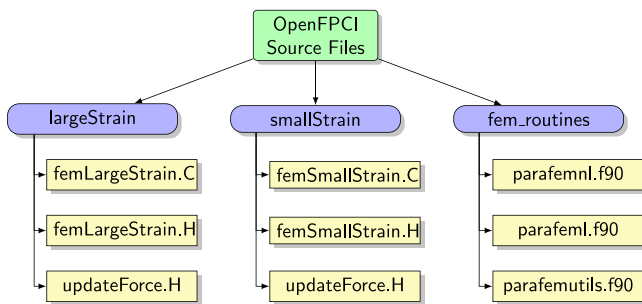
Fig. 3. OpenFPCI source file structure. The directories are highlighted in blue and files in yellow. The Fortran files are grouped in a single directory for convenience. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Fig. 4. Example of the compilation and linking process for an OpenFPCI plugin solving a large strain problem.
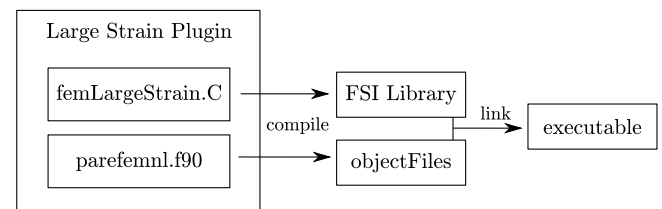
The relaxation factor ω can be a fixed value or dynamically changed based on Aitken [40] or IQN-ILS [41] relaxation methods.

3.2. OpenFPCI: Summary

OpenFPCI is summarised as a series of plugins, with each plugin solving a different engineering problem. The plugins, along with the FSI library link the OpenFOAM-Extend functionality with that of ParaFEM. A general diagrammatic summary of the implementation of an OpenFPCI plugin is provided in Fig. 1, with the following subsection providing further details. Before describing the development and structure behind these plugins, a summary of the major files within the source directory of OpenFPCI is presented.

The OpenFPCI source directory contains a number of directories (blue) and files (yellow), that are shown in Fig. 3. *fem_routines* holds all the Fortran files that are named with the prefix *parafem* and suffixed with their purpose. The Fortran files are separated into solvers and utilities. The utilities file is available to provide extra functionality, whilst the solver files contain the primary subroutines that are used to solve the engineering problems. For example, the file to solve the non-linear deformation of a structure is named *parafemnl.f90*.

The two folders (*largeStrain* and *smallStrain*) contain the class descriptions that are derived from the *solidsolver* base class, described briefly in Section 3.1. These classes act as wrappers around appropriate Fortran files, so that the ParaFEM subroutines can be called in a consistent manner from within the FSI library. For example, the *largeStrain* folder contains the class implementation solving the non-linear deformation of a structure and is a

wrapper around the subroutines found in *parafemnl.f90*. The files contained within these directories use a similar naming structure to the Fortran routines, prefixed with *fem* and suffixed with their purpose.

An OpenFPCI plugin is defined as the C++ wrapper/derived class along with its relevant Fortran solver. An example of the large strain plugin and the general process by which it is compiled and linked is shown in Fig. 4, with a comprehensive summary and description of each file within OpenFPCI provided in Table 1.

The C++ derived classes contain a number of methods, however two methods are utilised to call OpenFPCI subroutines. These methods have been highlighted in Fig. 2, the constructor of the class, *solid::solid()* and the call to solve the engineering problem *solid::evolve()*. These methods act as wrappers for the two subroutines that exist within the Fortran solver files, initialise and run. These subroutines are named *init* and *run* with their solver name suffixed i.e. within the non-linear solver they are named *initnl* and *runnl*. The two methods and subroutines are described extensively in Section 3.3, however a reduced example of the *femLargeStrain.C* file is provided in Listing 1, highlighting the declaration of external Fortran subroutines and the method calls that wrap these subroutines.

Listing 1: Reduced pseudocode of femLargeStrain.C, highlighting the approach taken for Fortran calls to *initnl* and *runnl* subroutines

```
// - Non-Linear Solver Subroutines
extern"C"{
  void initnl_()
  void runnl_()
}
// - Constructor
solid::solid(){
  initnl_()
}
// - Solve Structure
solid::evolve(){
```

Table 2
Inputs, and their purpose, to the OpenFPCI init subroutine.

Input	Purpose
g_coord	Global array of nodal coordinates
rest	Global restraints array
g_num_pp	Distributed element steering array
nn	Total number of nodes
nr	Total number of restrained nodes
Output	Purpose
g_g_pp	Distributed equation steering array
g_coord_pp	Distributed nodal coordinates

```
  include "updateForce.H"
  runnl_()
}
```

Using this wrapper isolates users from each side of the physical problem. Researchers can develop extensive run routines within ParaFEM without ever considering the OpenFOAM-Extend side of the problem.

3.3. OpenFPCI: init and run subroutines

3.3.1. Initialisation

The initialise and run subroutines are consistent across the plugins, and so will be referred to as *init* and *run* throughout the rest of this paper. The *init* subroutine has two major purposes: to read in the mesh information, the boundary conditions and generate the communication and steering arrays that are required by ParaFEM during the run phase. This section will highlight the processes that occur within the constructor of the class and the *init* subroutine. Table 2, provides a summary of the major inputs and outputs and their purposes. The remains of this section will provide a description of these variables and how they are generated within the class constructor.

Mesh information. ParaFEM mini-apps use a file based I/O, this means at the start of a mini-app all the mesh information is read in through files and any output data is written out to file. In essence a OpenFPCI plugin removes the input and output phases and replaces it with a direct memory transfer to and from OpenFOAM-Extend.

The constructor and *init* subroutine involves the conversion of an OpenFOAM-Extend mesh into a format that can be read and used by ParaFEM. The key arrays required by ParaFEM are a nodal steering array for each element and the coordinates of these nodes. The OpenFOAM-Extend mesh object can return these values however in order to pass them into a Fortran subroutine, they are copied into an array of primitive types. Listing 2 highlights the copying process, within *solid::solid()*, of the mesh coordinates and steering array, that are passed into *init*.

Listing 2: Pseudocode of the copying process of the steering array and coordinates of the mesh from OpenFOAM's mesh object to a one dimensional array, to pass into the *init* fortran routine

```
// — Coordinates
// — Create coordinates field from IOobject.
pointIOField coord
(
  IOobject(...)
)

// — For each point in mesh
for(int i=0; i < totalPoints ; i++){
  g_coord[index++]  =  coord[i].x();
  g_coord[index++]  =  coord[i].y();
  g_coord[index++]  =  coord[i].z();
}
```

```
// — Steering Matrix
//— For each cell in mesh
for(int cell=0; cell < totalCells; cell++){

  // Current cells steering array
  labelList& curCell = mesh.cellPoints()[cell];

  // — For each node in current cell
  for(int node=0; node < nodesPerCell; i++){

      // — Copy into vector and +1 for Fortran
      g_num_pp[index++] = curCell()[node]+1;
  }
}
```

A *pointIOField* containing the nodal coordinates of the mesh is stored in OpenFOAM-Extends *objectRegistry*, and can be read in using an *IOobject*. This field, *coord*, is subsequently copied into *g_coord*, the global coordinate array, for ParaFEM. The nodal steering array can be accessed through the *mesh.cellPoints()* method, and is copied into *g_num_pp*. Variables using the *_pp* suffix imply that they are stored on a per MPI process basis, so in parallel each MPI process will only hold the steering array for its local elements (associated with the local subdomain).

The order of node numbers stored for each element (element steering) in OpenFOAM is different from the node ordering convention used in ParaFEM, so conversion from one convention to the other is required. Within the *init* subroutine, a subroutine, *of2sg* is used to convert the mesh formats. This subroutine exists within the *parafemutils.f90*. The process loops over each element, taking in the element type and its current element steering array and returning the updated one. The subroutine uses a Fortran SWITCH statement and so implementing differing element types can easily be done by adding the element name and its conversion.

Initial and boundary conditions. As with the mesh generation, the initial and boundary conditions are transferred to an OpenFPCI solid mechanics plugin through a similar process. The *0* folder within OpenFOAM-Extend case files contains the initial conditions. A pointVectorField dictionary named pointD is initialised within the folder, this file contains the nodal displacements in relation to the starting coordinate system. Within this file patches can be defined as set types, the task is to transfer these types into a format that can be passed to ParaFEM. The *rest* array is populated with the first column containing the node and the preceding columns the x, y and z restraints, 1 — fixed and 0 — free. Listing 3 shows the loop and the condition required for a completely fixed restraint.

Listing 3: Example loop within femLargeStrain to allocate the restrained nodes for a fixed boundary condition

```
// — Loop through each patch
for(int patch=0; patch < totalPatches; patch++)
{
  // — If the patch has a fixedValue type
  if
  (
    isA<fixedValuePointPatchVectorField>
    (
      pointD_.boundaryField()[patch]
    )
  )
  {
    // Create list of nodes
    // Set x=y=z=0 for each node
  }
}
```

Within the listing, the code creating the list of nodes and restraints is removed. The mesh domain decomposition methods available within OpenFOAM-Extend can be used when decomposing the mesh. In parallel each MPI process creates a list of

Table 3
Inputs and outputs to the run subroutine for the large strain OpenFPCI plugin.

Input	Purpose
nodes	List of nodes with external forces
val	Array of forces associated with the *node* list
num_var	Numerical variables, i.e. time step
mat_prop	Material properties, i.e. Young's Modulus
nr	Total number of restrained nodes
nf	Total number of loaded nodes
gravlo_pp	Distributed body/gravity loads
g_g_pp	Distributed equation steering array
g_coord_pp	Distributed nodal coordinates
g_num_pp	Distributed element steering array

In/Output	Purpose
Dfield	Distributed displacement field
Ufield	Distributed velocity field
Afield	Distributed acceleration field

restrained nodes that exist in its sub domain, all the lists are subsequently gathered into a global list and cleaned. This cleaning process involves removing any duplicated nodes. A node may exist on multiple patches and multiple processors. If it exists on multiple processors then the duplicates are removed. If the node exists on multiple patches it may have a different restraint. In this situation the nodes that are most fixed are kept and the rest removed from the list. This list is then scattered to the rest of the processors and copied into the *rest* array, as shown in Listing 4.

Listing 4: The loop copying the restrained list into the rest array that is passed to ParaFEM

```
for(int listI=0; listI < masterRest.size(); listI++){
  // Copy List
  rest_[ nr * 0 + index ] =  RestrainedList[listI][0];
  rest_[ nr * 1 + index ] =  RestrainedList[listI][1];
  rest_[ nr * 2 + index ] =  RestrainedList[listI][2];
  rest_[ nr * 3 + index ] =  RestrainedList[listI][3];
  index++;
}
```

Finally a list of nodes that is externally loaded is generated. Within the *0* case folder the volVectorField *D*, represents the elemental displacements. Within this file the type can be defined as *tractionDisplacement*, which specifies that the nodes on this patch are externally loaded. The same methodology that is used to generate the *rest* array, is used to create a list of nodes, *forcedNodes*.

3.4. Run

Once all the data exchange house keeping tasks have been completed during the initialisation process, the run subroutine is used to solve the engineering problem. This takes the accelerations, velocities and displacements as inputs, updates them based on the new solution and passes them back out as outputs to OpenFOAM-Extend. The subroutine solves Eq. (1). Fig. 5 shows the reduced pseudocode of the *run* subroutine for the non-linear OpenFPCI plugin with a list of inputs and outputs shown in Table 3.

The run routine begins by loading the structure with the external forces. The external forces are provided in two arrays, a list of nodes, *node* and their vector values, *val*. These are converted to an external force array in its equation format using the ParaFEM subroutine *Load*. This subroutine uses *node*, *val* and the equation steering matrix *g_g_pp* as inputs and outputs *f_ext_pp*. The displacements, velocities and accelerations are input in an array arranged by nodes and elements. These are converted into an array of values in an equation format using the *scatter* subroutine in ParaFEM. The system of equations to be solved is given
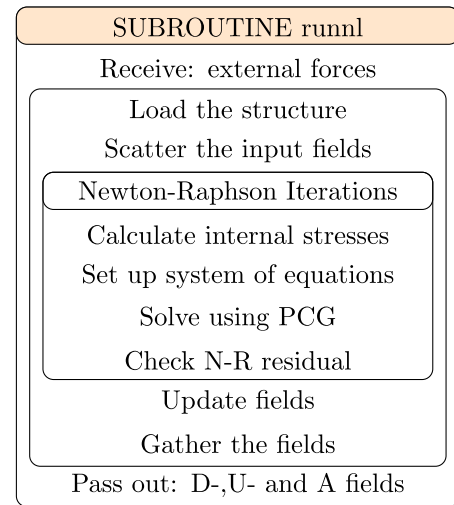
Fig. 5. Pseudocode for the run subroutine of the large strain OpenFPCI plugin.

by Eq. (12), following the solution procedure for geometrically non-linear materials given in Bathe [42].

$$\{\mathbf{r}\_pp\} = \{\mathbf{f}\_ext\_pp\} - \{\mathbf{f}\_int\_pp\} + [\hat{\mathbf{K}}\_pp]\{\hat{\mathbf{x}}\_pp\}, \qquad (12)$$

where $\{\mathbf{r}\_pp\}$ is the residual vector, $\{\mathbf{f}\_ext\_pp\}$ the external force vector, $\{\mathbf{f}\_int\_pp\}$ the internal force vector, $[\hat{\mathbf{K}}\_pp]$ the effective stiffness matrix and $\{\hat{\mathbf{x}}\_pp\}$ the effective displacement vector. This system of equations is solved during each Newton–Raphson iteration using the parallelised element-by-element PCG method. Once the difference between two successive N-R iterations is small enough the displacement, velocity and accelerations are updated and scattered from their equation array to an array arranged by elements and nodes, ready to be passed back to OpenFOAM-Extend.
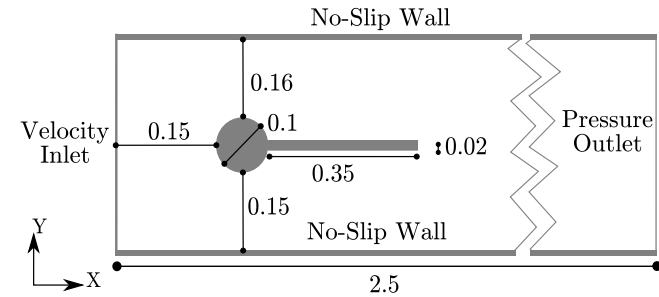
4. Validation

The validation of the application is based on a well-known benchmark developed by Turek and Hron [43]. This is a well-defined suite of tests to validate both the structural solver and fluid solver independently, as well as fully coupled FSI tests. Within the reference paper, the suites of tests are referred to as CSM 1–3, CFD 1–3 and FSI 1–3. The reference paper utilises a fully implicit monolithic finite element approach with ALE and shows strong convergence of results for the suite of tests. The authors carried out the independent solver validation tests, CSM 3 and CFD 3, using ParaFEM and OpenFOAM-Extend respectively. The results compare well with the benchmark, which is to be expected as both software packages are well established and professionally maintained. The structural solver and fluid solver benchmark results are not provided here. The FSI benchmark, FSI 3, that is presented for the validation of the coupled problem, involves the two-dimensional laminar flow of an incompressible Newtonian fluid around a rigid cylinder with an elastic flag attached behind. A schematic of the computational domain is provided in Fig. 6 with the properties of both the solid and fluid provided in Table 4.

The subscripts, $_f$ and $_s$ represent the fluid and solid properties respectively, with v_s representing the Poisson ratio and E the material stiffness. Three dimensionless numbers are presented. The Mass number, M, the Reynolds number, Re and the inverse of the Cauchy number, Ae, that relates the inertial and the elastic forces within the problem. The outlet has a fixed pressure, set to zero, with the rest of the walls having a zero gradient pressure

Table 4

Dimensionless numbers and the material properties of the fluid and solid domains.

Property	Value
ρ_f [kg m^{-3}]	1000
v_f [m^2 s^{-1}]	0.001
$\bar{u}$ [m s^{-1}]	2
ρ_s [kg m^{-3}]	1000
v_s	0.4
E [kg m^{-1} s^{-2}]	5.4e6
$M = \frac{\rho_s}{\rho_f}$	1
$Re = \frac{\bar{u}D}{v_f}$	200
$Ae = \frac{E}{\rho_f \bar{u}^2}$	5.4e3

Fig. 6. Computational domain for the FSI benchmark test case.

Table 5

Displacements at the tip of the elastic flag for the FSI benchmark.

Grid	$d_x[10^{-3}$m] Mean $\pm$Amplitude	$d_y[10^{-3}$m] Mean $\pm$Amplitude
1	−1.95 ±1.85	1.84 ±27.9
2	−2.62 ±2.48	1.54 ±33.1
3	−2.88 ±2.73	1.47 ±34.9
4	−2.98 ±2.81	1.45 ±35.5
[44]	−2.88 ±2.72	1.47 ±35.0

Table 6

Lift and drag forces over the cylinder and elastic flag for the FSI benchmark.

Grid	Drag(N) Mean $\pm$Amplitude	Lift(N) Mean $\pm$Amplitude
1	455.7 ±16.6	2.13 ±146.4
2	458.8 ±23.5	2.63 ±163.6
3	460.7 ±27.2	2.67 ±173.0
4	461.3 ±28.4	2.62 ±176.3
[44]	460.5 ±27.7	2.50 ±153.9

condition. The walls above and below the structure are considered no-slip, with the inlet having a parabolic velocity profile described by Eq. (13). The walls normal to the z-direction are considered empty, ignoring the z-component within the governing equations.

$$u(y) = 1.5 \times \bar{u} \times \frac{y(H - y)}{(\frac{H}{2})^2} \qquad (13)$$

Three meshes were used to check for convergence of the results. The solid domain mesh is fixed at 3000 elements whilst four fluid meshes labelled: Grid-1 (∼5000), Grid-2 (∼21,000), Grid-3 (∼85,000) and Grid-4 (∼340,000), are used. Fig. 7 provides an example of the grid used for the grid convergence study. The fluid mesh is from Grid-2 and the solid mesh is that used for all the grids.

The fluid domain uses a block structured mesh with hexahedral control volumes and the solid domain is made up of linear eight noded hexahedral elements. The problem is two dimensional and so the nodal freedoms in the z direction are fixed for each element, and the elements in contact with the cylinder are fixed in the x, y and z directions. ParaFEM only has a limited number of two dimensional elements (as two dimensional problems typically do not require solution on supercomputers) and so three dimensional elements are used in this test case. The Newmark-beta constants used to advance the solid solution in

time are, $\beta = 0.4$ and $\delta = 0.6$. The FSI tolerance was set at, *tol* = 1e−9, and used the IQN-ILS relaxation scheme [41].

The elastic flag begins in its undeformed configuration and is held in this position for two seconds, to allow the flow to develop over the structure. Once two seconds has been reached, the beam is released and is allowed to deflect under the pressure and viscous loading of the structure. Within three to four seconds the simulation has reached a periodic state. Fig. 8 provides an example of the velocity and displacement field within the domain at time = 8.5 s.

The quantities for comparison with the reference case are the displacement in the X and Y directions at the tip of the beam, undergoing an oscillatory motion, and the lift and drag forces over the cylinder and elastic beam. The profiles for the displacement can be seen in Figs. 9a and 9b, with Table 5 highlighting the mean and amplitude of the displacement. The lift and drag forces are subsequently shown in Figs. 10b and 10a and the mean and amplitude values are shown in Table 6.

From the results for displacement, it can be seen the maximum and minimum values of displacement in the X direction are similar to those seen in the reference, however the amplitude in the Y direction is marginally higher than the reference case in both the positive and negative directions. The trends seen in the displacement fields are also observed in the approximations for lift and drag. The mean drag converges to the results in the reference paper, with a 0.017% error and the amplitude has an error of 2.6%. The frequency of lift compares well, with the value calculated (5.50 Hz) having approximately a 0.7% error compared with the reference paper (5.46 Hz). Within the literature a wide spread of results are reported for this benchmark test case, and are summarised in a table by Turek et al. [44]. With the clear convergence of results and the small percentage error in the frequency of lift in comparison to the reference, the authors are satisfied that the application is performing in the correct manner for FSI computations.
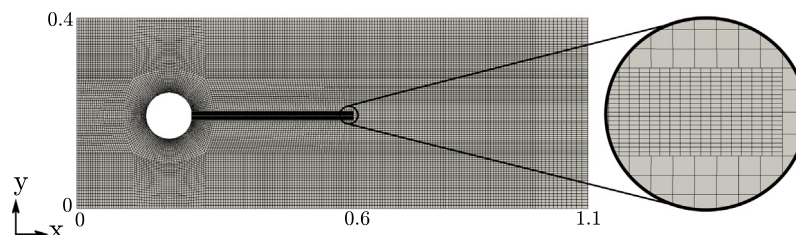
Fig. 7. Example of the fluid mesh (Grid 2) and solid mesh used for the grid convergence study.
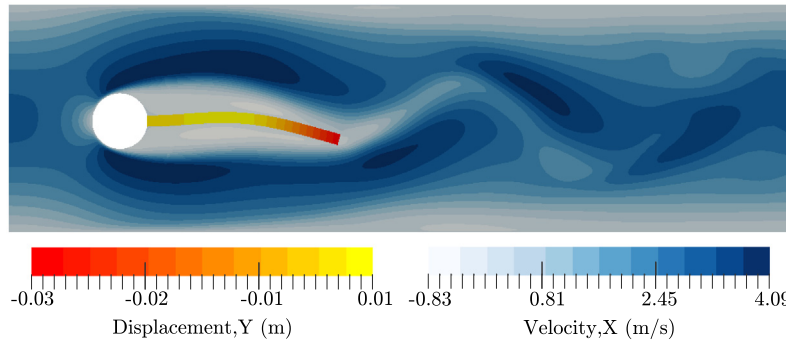
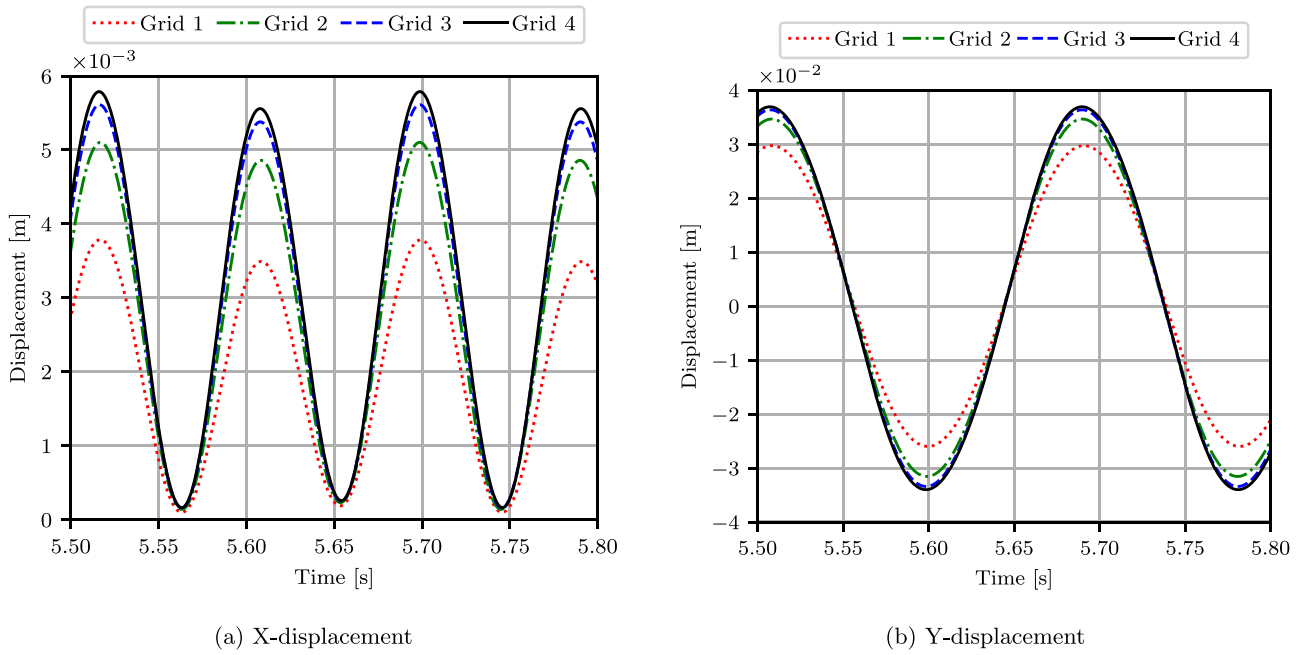**Fig. 8.** Displacement and velocity field at time = 8.5 s.

(a) X-displacement

(b) Y-displacement

Fig. 9. X and Y displacement at the tip of the elastic flag.
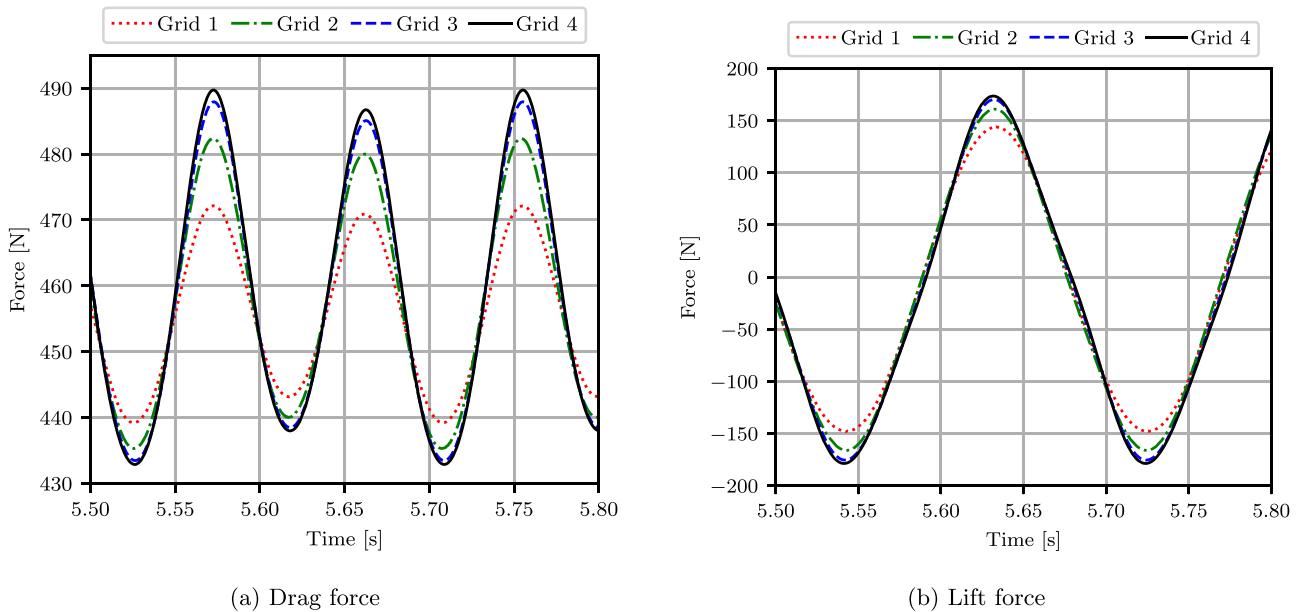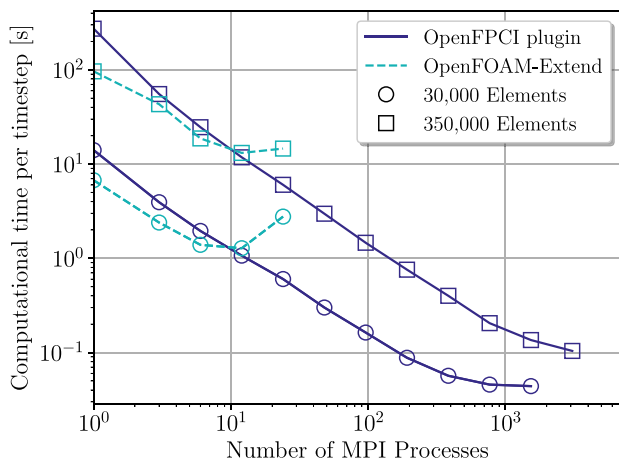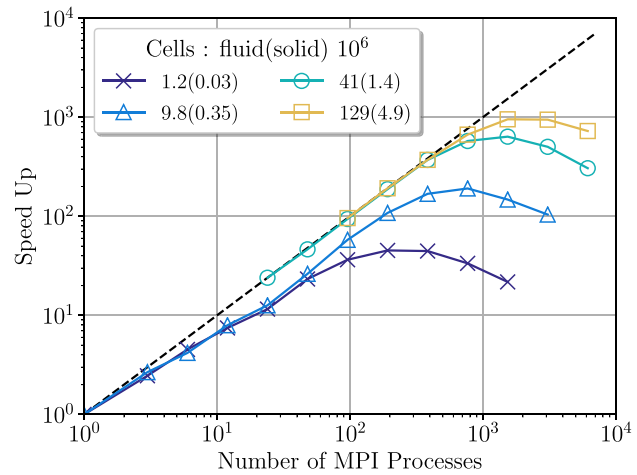

(a) Drag force

(b) Lift force

Fig. 10. Lift and drag forces for FSI benchmark.

Fig. 11. (a) A comparison of the large strain OpenFPCI plugin and the total Lagrangian solver available within OpenFOAM-Extend (b) The speed up of the full application running for 30 time steps across a series of mesh sizes.

5. Parallel performance

Both applications make use of the Message Passing Interface (MPI) and can be run on both shared memory and distributed architectures, with OpenFOAM-Extend using pre processing domain decomposition and ParaFEM using runtime element by element techniques. A range of domain decomposition techniques are available within OpenFOAM-Extend, OpenFPCI makes use of these domain decomposition methods to decompose the structural mesh, as a preprocessing step.

Large single physics libraries and packages usually wrap the underlying MPI calls, in OpenFOAM this is through the Pstream class and in ParaFEM an mp_interface module. With both of these libraries the MPI calls are based on the world communicator, MPI_COMM_WORLD, and so without significant alteration of these libraries the use of differing communicators is difficult. Therefore the current version of OpenFPCI utilises the same number of cores for the solid and fluid domains. The coupling environments CoMa and MpCCI use a central server to communicate all data, OpenFPCI uses a similar approach with the interface mapping and data handling being done by the master processor.

Two scalability studies are shown to highlight both the overall scalability of the application when using an OpenFPCI plugin and a study comparing the scalability of an OpenFPCI plugin with OpenFOAM-Extend's standard non-linear structural solver. All the test were preformed on the Tianhe2 machine at the Chinese Supercomputing Centre in Guangzhou [45]. Each node compromises two Intel Ivy Bridge Xeon E5-2692 CPU's running at clock speed of 2.2 GHz. OpenFOAM-Extend, ParaFEM and OpenFPCI were all compiled with the latest Intel 17.0.6 compilers, with MPICH version 3.2.1 used for parallel processing.

Fig. 11 highlights a number of benefits and current limitations of the application. Firstly Fig. 11a highlights the benefits of using OpenFPCI over the current implementation available within OpenFOAM-Extend. The figure compares the execution time for a single solution using the large strain OpenFPCI plugin and the large strain structural solver available within OpenFOAM-Extend, for two cases. These cases contain approximately 30,000 and 350,000 elements. The large strain OpenFPCI plugin scales almost ideally to around 1000 MPI processes for a problem with approximately 30,000 elements and 3000 MPI processes for the case involving 350,000 elements. In comparison the OpenFOAM-Extend solver scales to around 12 cores only and achieves very

Table 7
Comparison of the OpenFPCI and OpenFOAM-Extend FSI simulation results, against the reference paper.

	$d_x[10^{-3}\text{m}]$	$d_y[10^{-3}\text{m}]$	Drag(N)	Lift(N)
	Mean ±Amplitude			
OpenFPCI	−2.98 ±2.81	1.45 ±35.5	461.3 ±28.4	2.62 ±176.3
OpenFOAM-Extend	−2.96 ±2.80	1.46 ±35.5	460.7 ±28.6	2.32 ±162.6
[44]	−2.88 ±2.72	1.47 ±35.0	460.5 ±27.7	2.50 ±153.9

little improvement after this for either case. The OpenFOAM-Extend solver did however preform better for smaller core counts, below 12, with improved execution times over the OpenFPCI plugin. Table 7 provides a comparison of the results achieved by the OpenFOAM-Extend structural solver, OpenFPCI and the reference, for clarification.

The results for both the OpenFOAM-Extend solvers were completed using grid 4, and have less than 1% difference between the OpenFPCI case for the mean and amplitude of displacement in the *x*-direction and *y*-direction and the drag force.

The strong scalability study for the full application is shown in Fig. 11b. It considered the overall scalability of the application for 30 time steps, and can be seen to scale well to around 3072 MPI processes before the performance decreases. The overall scalability of the application is a combination of the scalability of each of the processes shown in the FSI algorithm, Fig. 2.
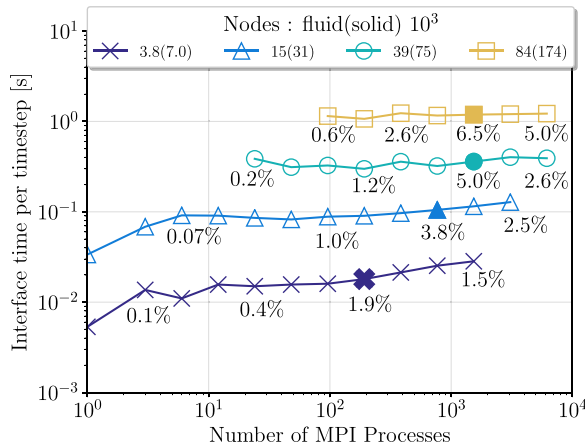
Fig. 12 and Table 8 provide some further insight into the effects of the interface and the overall breakdown of time spent in each region.

The interface of the application uses a single processor to deal with the data mapping and interface updates. It can be seen from Fig. 12 that the time taken in the interface for each mesh is generally consistent across the MPI processes, and that as the number of nodes at the interface grows so does the time taken for all the work to be completed. The simulations taking the shortest time are highlighted for each mesh, with the percentage of time in the interface shown for a number of the results. The percentage of time in the interface peaks at the highlighted value, where the overall execution time of the other processes is at a minimum. In all the meshes it was seen that OpenFOAM-Extend no longer scales after this point, with the execution time staying the same or increasing for both solving the governing equations and moving the fluid mesh. Table 8 shows the time taken, per time step, for each of the four major processes in Fig. 2, of the

Table 8
Computational time for each process for the overall fastest simulation at that mesh size, highlighted in Fig. 12.

Mesh	Computational time to solve each process, per timestep [s]			
	Solid	Fluid	Fluid mesh	Interface
✕	0.09	0.41	0.4	0.02
△	0.2	1.0	1.4	0.1
○	0.45	2.63	2.5	0.4
▢	1.6	7.1	8.5	1.2

Fig. 12. The computational time spent within the interface per timestep for four meshes with different numbers of nodes at the interface. The percentage of time spent within the interface is annotated for a number of the simulations, with the quickest overall simulation highlighted in bold.

quickest results for each mesh, also highlighted by a bold marker in Fig. 12. It can be seen from the table that the time spent solving the solid is small in comparison to that of moving the mesh and solving the fluid governing equations. This suggests the overall scalability of the current problem with mesh ratios of around 25–40 fluid elements to one solid element is limited by the scalability of fluid domain. The moving of the mesh and the solving of the fluid governing equations, scale differently. Solving the Laplacian of the boundary velocity to smooth the mesh, does not scale well, to the same number of cores as solving the governing equations. A balance between the two must be found.

To summarise the results, if users only have access to desktop systems with a maximum of 12 cores, they will see very little benefit from using OpenFPCI as opposed to capabilities already available in OpenFOAM-Extend. However for users looking to increase the size of the problem within the solid domain and that have access to larger systems, significant benefits can be seen. For mesh sizes in tens of thousands a factor of 30 speed up is seen and for meshes in the hundreds of thousands a factor of approximately 125 can be achieved. Although the speed up factor can be seen when comparing the solid solvers available, the overall speedup will heavily depend on the overall time impact of the solid itself. For problems with an approximately 25–40:1 fluid to solid mesh ratio good speed up can be achieved up to around 3072 cores. The interface approach, which is similar to a number of general coupling environments [18,21], stores all the interface information on the master processor where data mapping occurs. Once the data mapping between the fluid and solid domains is complete the information is distributed to the other cores. Developing this interface approach to work in parallel and utilising different MPI communicators between OpenFOAM-Extend and ParaFEM would provide further speed up, however

to the authors knowledge the level of scaling shown in the above section is rare amongst current FSI applications.

6. Conclusions

This paper has presented OpenFPCI, an open source FOAM to ParaFEM coupling interface. OpenFPCI has been developed to couple the extensive fluid dynamics capabilities of OpenFOAM-Extend with the highly parallel nature of ParaFEM for solving structural engineering problems. ParaFEM is structured as a series of mini-apps with each mini-app solving a particular engineering problem. OpenFPCI is a series of plugins, with each plugin being developed from a particular ParaFEM mini-app and solving a specific problem. The plugins decompose a ParaFEM executable program into a series of subroutines to initialise and solve the engineering problem. These subroutines are called by OpenFOAM-Extend in a master–slave approach. The focus has been on defining an OpenFPCI plugin that can be used to solve FSI problems where the structure is modelled as a St. Venant–Kirchhoff material.

After identifying good agreement with a benchmark validation case the parallel scalability of the plugin was compared directly against the structural solver available within OpenFOAM-Extend. It was discovered that the OpenFPCI plugin provided significant execution time improvements over the OpenFOAM-Extend solver. Although the OpenFPCI plugin was slower up to around 12 cores, it continued to scale onto core counts in the thousands that the OpenFOAM-Extend solver did not. For solid meshes in the tens of thousands, a speed up improvement of approximately 30 was achieved, and for meshes in the hundreds of thousands, a speed up factor of 125 was observed. For users with access to large HPC systems with core counts ranging from 20 to 3000 the use of OpenFPCI can provide significantly improved solution times in comparison to OpenFOAM-Extend. The overall scalability of the OpenFPCI plugin within a full FSI simulation was shown on a series of test cases up to 130 million fluid cells and 5 million solid elements, where almost ideal scaling was seen on up to 1536 cores.

Both applications involved with in the coupling are open source; OpenFOAM-Extend is released under GPL and ParaFEM via BSD licence. OpenFPCI is therefore released under a BSD licence and is freely available at its public repository, OpenFPCI, where all of the source code is released and a number of example problems are provided including the validation case shown. The current release is stable and major improvements will be released as a separate entity so that issues with backward compatibility can be mitigated. It is hoped that this paper will encourage the use of current OpenFPCI plugins and the development of further capabilities.

Appendix A. Framework extension

One of the goals of this paper is to encourage the development of other plugins that can be used within the FSI library. A series of template files are provided that enable developers to implement their own plugins. These template files can be copied and renamed along with the naming standards described at the start of this section. A ParaFEM mini-app should be selected that either solves the structural problem at hand or is close enough that implementation of the added functionality is simple. The program should be decomposed into the initialisation phase and solution phase, which within ParaFEM programs are clearly labelled. These can then be exported into the *init** and *run** Fortran subroutines within the *parafem*.f90* file created. If additional parameters are required to be read from OpenFOAM dictionaries they can be added to the *fem** C and H files in a similar manner to the numerical parameters. There are a number of additional useful subroutines within the *parafemutils.f90* file, with the list of subroutines documented within the source code. A summarised step by step guide to implementing a new plugin is provided below.

Step 1: Locate a ParaFEM mini-app that is most appropriate.

Step 2: Decompose into an **Initialisation** and **Solution** phase.

Step 3: Copy template files for the ParaFEM routines and derived classes and rename them appropriately, *parafem*.f90* and *fem*.C/H*.

Step 4: Port the code into the *init** and *run** within the *parafem*.f90*.

Step 5: Add any additional inputs and outputs to the Fortran subroutines, remembering to update the Extern C definitions within the *fem*.C* file. Examples of reading in numerical parameters and creating new fields from the large strain OpenFPCI plugin are shown.

Numerical Parameters can be read in through OpenFOAM dictionaries, for example reading in the beta parameter for the Newmark scheme.

```
double beta (readScalar(solidProperties().lookup("
    beta")));
```

Fields can be read in through OpenFOAM-Extend IOobjects, an example of creating a nodal displacement field.

```
pointD_
(
  IOobject
  (
    "pointD",
    runTime().timeName(),
    mesh,
    IOobject::READ_IF_PRESENT,
    IOobject::AUTO_WRITE
  ),
  pMesh_
)
```

In general, for most plugins the *init* phase will be similar, it will read in the mesh and create the necessary ParaFEM arrays. It will be the *run* subroutine that differs most significantly.

Appendix B. Downloading and installing OpenFPCI

The installation of OpenFPCI requires that both OpenFOAM-Extend and ParaFEM are compiled on the system, using the OpenMPI available on the system. The instructions to download and install can be found at their respective websites, OpenFOAM-Extend and ParaFEM. The latest version of OpenFPCI v1.1 can be downloaded from the git repository OpenFPCI. The repository also contains more detailed installation instructions. Installation is completed through the use of a bash script in the src directory *src/openfpci.sh*. The paths to the ParaFEM home directory and OpenFOAM-Extend home directory are required before the script is run.

Listing 5: Installation instructions for OpenFPCI

```
echo "export PARAFEM_DIR=path/to/parafem code/parafem" >> ~/.
    bashrc
echo "export FOAM_DIR=path/to/foam/foam extend.x.x" >> ~/.
    bashrc
source ~/.bashrc
cd OpenFPCI/src
./openfpci.sh
```

The script downloads and compiles the FSI library required before linking the files required for OpenFPCI. The software has been tested using OpenFOAM-Extend 4.0 and ParaFEM.5.0.3 on a Linux workstation running Ubuntu 16.04 and OpenMPI 1.6.5. The application has been installed on the ARCHER [46] high performance computing system, the UK's national supercomputing service and Tianhe2, at the Chinese National Supercomputing Centre in Guangzhou [45].

Appendix C. Using OpenFPCI

The Foam-Extend FSI library [17] is accompanied by a series of test problems. These test problems have been included within the OpenFPCI repository with any additional files and input parameters included. A test case directory contains two folders, a fluid and solid folder that contain information about each of the domains properties and mesh. A diagrammatic view of the file structure is provided in Fig. C.13:

The fluid case files and solid case files contain the information required for fluid and solid domains respectively. A soft link is created between the major files within the two directories to couple the two domains. A case is subsequently run in the same manner as a standard OpenFOAM case, by entering into the fluid case folder and running the executable. Fig. C.13 highlights a number of files. Those files highlighted in green are the nodal quantities of displacement, velocity and acceleration, and those in blue are the case files used to alter OpenFPCI's material and numerical properties. An example of the solidProperties file is provided in Listing 6.

Listing 6: Numerical parameters for the large strain OpenFPCI plugin

```
// Solver type
solidSolver femLargeStrain;

femLargeStrainCoeffs
{
    // Gravity Loading
    gravity 0.0;
    // Newmark Parameters
    beta    0.25;
    delta   0.5;
    // RBF Interpolation
    rbf     no;
}
```
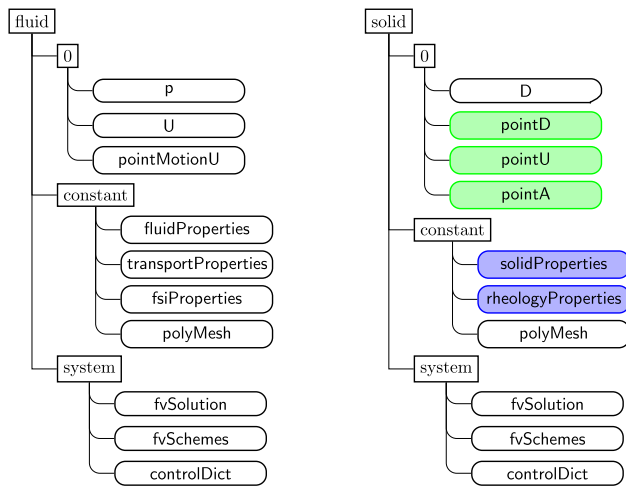
Fig. C.13. Case setup for a problem, with additional fields required for OpenFPCI highlighted in green and files from which parameters, required for OpenFPCI, are read in blue. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The OpenFPCI solver selected is that which follows *solidSolver*, the coefficients for this solver must then be complete or the case will fail to run. As in the example the large strain solver, *femLargeStrainCoeffs*, uses the Newmark time stepping scheme and so requires values for beta and delta.

Running cases in parallel works in a similar manner to a standard OpenFOAM case. The *decomposePar* utility can be used to decompose the mesh. It uses the *solid/system/decomposeParDict* dictionary that defines the number of processes and decomposition method used. This process creates a series of directories named *processor* followed by the processor number, starting at 0. A soft link is created between the processor directories within the solid folder and the processor directories of the decomposed fluid case. The current release requires the same number of processors to be used for each domain. Running in parallel then uses the MPI runtime command, executed from inside the fluid case folder. An example of running the command on 48 cores is as follows:

mpirun -np 48 fsiFoam -parallel

ParaFEM can output results files in ensi gold format to be viewed in the visualisation package ParaView, however when writing the results files for parallel computations, the current methodology gathers all the data to the master processor that subsequently writes the data to file. This is effective for steady or static problems where only the final result is required, however to output data at multiple time steps this method can be time consuming. The quantities of interest are passed between the OpenFOAM and ParaFEM and can be written out using Open-FOAM's parallel I/O capabilities. Each processor writes a time file containing the fields for that decomposed area of mesh. These files can be reconstructed into a full domain using the *reconstructPar* utility and ParaView can be used to visualise the results. OpenFOAM provides a wrapper around ParaView, and so the *paraFoam* command can be used from within the fluid case folder to view both the fluid and solid domains.

References

[1] C.M. Scotti, E.A. Finol, Comput. Struct. 85 (11–14) (2007) 1097–1113, http://dx.doi.org/10.1016/j.compstruc.2006.08.041.

[2] J. Hron, M. Mádlík, Nonlinear Anal. RWA 8 (5) (2007) 1431–1458, http://dx.doi.org/10.1016/j.nonrwa.2006.05.007.

[3] B. Owen, N. Bojdo, A. Jivkov, B. Keavney, A. Revell, Biomech. Model. Mechanobiol. (2018) 1–26, http://dx.doi.org/10.1007/s10237-018-1024-9.

[4] M.-C. Hsu, Y. Bazilevs, Comput. Mech. 50 (6) (2012) 821–833, http://dx.doi.org/10.1007/s00466-012-0772-0.

[5] A. Korobenko, J. Yan, S. Gohari, S. Sarkar, Y. Bazilevs, Comput. & Fluids 158 (2017) 167–175, http://dx.doi.org/10.1016/j.compfluid.2017.05.010.

[6] C. Gebhardt, B. Roccia, Renew. Energy 66 (2014) 495–514, http://dx.doi.org/10.1016/j.renene.2013.12.040.

[7] S. Hewitt, L. Margetts, A. Revell, Archives of Computational Methods in Engineering, Springer Netherlands, 2017, pp. 1–21, http://dx.doi.org/10.1007/s11831-017-9222-7.

[8] C. Michler, S.J. Hulshoff, E.H. van Brummelen, R. de Borst, Comput. & Fluids 33 (5–6) (2004) 839–848, http://dx.doi.org/10.1016/j.compfluid.2003.06.006.

[9] H.G. Matthies, J. Steindorf, Comput. Struct. 81 (8–11) (2003) 805–812, http://dx.doi.org/10.1016/S0045-7949(02)00409-1.

[10] M. Heil, A.L. Hazel, J. Boyle, Comput. Mech. 43 (1) (2008) 91–101, http://dx.doi.org/10.1007/s00466-008-0270-6.

[11] ADINA: Fluid-Structure Interaction.

[12] ANSYS: Fluid-Structure Interaction.

[13] COMSOL Multiphysics: A Simulation Platform for Physics-Based Modeling.

[14] J. Lorentzon, Thesis, 2009, p. 86.

[15] F. Palacios, J. Alonso, K. Duraisamy, M. Colonno, J. Hicken, A. Aranake, A. Campos, S. Copeland, T. Economon, A. Lonkar, T. Lukaczyk, T. Taylor, 51st AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition, American Institute of Aeronautics and Astronautics, Reston, Virigina, 2013, http://dx.doi.org/10.2514/6.2013-287.

[16] A. Cesur, C. Carlsson, A. Feymark, L. Fuchs, J. Revstedt, Comput. & Fluids 101 (2014) 27–41, http://dx.doi.org/10.1016/j.compfluid.2014.05.012.

[17] Z. Tukovic, P. Cardiff, A. Karac, H. Jasak, A. Ivankovic, 9th OpenFOAM Workshop, Zagreb, 2014.

[18] T.G. Gallinger, Effiziente Algorithmen Zur Partitionierten LÖSung Stark Gekoppelter Probleme Der Fluid-Struktur-Wechselwirkung, Ph.D. thesis, The Technical University of Munich, 2010, p. 183.

[19] M. Breuer, G. De Nayer, M. Münsch, T. Gallinger, R. Wüchner, J. Fluids Struct. 29 (2012) 107–130, http://dx.doi.org/10.1016/j.jfluidstructs.2011.09.003.

[20] H.-J. Bungartz, F. Lindner, B. Gatzhammer, M. Mehl, K. Scheufele, A. Shukaev, B. Uekermann, Comput. & Fluids 141 (2016) 250–258, http://dx.doi.org/10.1016/j.compfluid.2016.04.003.

[21] MpCCI: A Multiphysics Coupling Environment.

[22] I.M. Smith, D.V. Griffiths, L. Margetts, Programming the Finite Element Method, fifth ed., John Wiley & Sons, Ltd, 2014.

[23] H.G. Weller, G. Tabor, H. Jasak, C. Fureby, Comput. Phys. 12 (6) (1998) 620, http://dx.doi.org/10.1063/1.168744.

[24] M.A. Heroux, D.W. Doerfler, P.S. Crozier, J.M. Willenbring, H.C. Edwards, A. Williams, M. Rajan, E.R. Keiter, H.K. Thornquist, R.W. Numrich, Improving Performance Via Mini-Applications, Tech. Rep. SAND2009-5574, Sandia National Laboratories (2009).

[25] I.M. Smith, L. Margetts, VII International Conference on Computational Plasticity, Barcelona, 2003.

[26] F. Levrero-Florencio, P. Pankaj, Front. Physiol. 9 (2018) 545, http://dx.doi.org/10.3389/fphys.2018.00545.

[27] A. Shterenlikht, L. Margetts, Proc. R. Soc. Lond. Ser. A Math. Phys. Eng. Sci. 471 (2177) (2015) http://dx.doi.org/10.1098/rspa.2015.0039.

[28] L. Evans, L. Margetts, V. Casalegno, L. Lever, J. Bushell, T. Lowe, A. Wallwork, P. Young, A. Lindemann, M. Schmidt, P. Mummery, Fusion Eng. Des. 100 (2015) 100–111, http://dx.doi.org/10.1016/j.fusengdes.2015.04.048.

[29] J.D. Arregui-Mena, L. Margetts, D.V. Griffiths, L. Lever, G. Hall, P.M. Mummery, J. Nucl. Mater. 465 (2015) 793–804, http://dx.doi.org/10.1016/j.jnucmat.2015.05.058.

[30] N.M. Newmark, J. Eng. Mech. Div. 85 (3) (1959) 67–-94.

[31] I.M. Smith, A. Wang, Int. J. Numer. Anal. Methods Geomech. 22 (10) (1998) 777–790, http://dx.doi.org/10.1002/(SICI)1096-9853(1998100)22:10<777::AID-NAG940>3.0.CO;2-U.

[32] I.M. Smith, L. Margetts, Eng. Comput. 23 (2) (2006) 154–165, http://dx.doi.org/10.1108/02644400610644522.

[33] H. Jasak, Z. Tukovic, European Conference on Computational Fluid Dynamics, 2010, pp. 1–19.

[34] R. Löhner, C. Yang, Commun. Numer. Methods. Eng. 12 (10) (1996) 599–608, http://dx.doi.org/10.1002/(SICI)1099-0887(199610)12:10<599::AID-CNM1>3.0.CO;2-Q.

[35] H. Jasak, Z. Tukovic, Transactions of Famena, Vol. 30, 2006, pp. 1–20.

[36] R. Issa, J. Comput. Phys. 62 (1) (1986) 40–65, http://dx.doi.org/10.1016/0021-9991(86)90099-9.

[37] J. Degroote, Arch. Comput. Methods Eng. 20 (3) (2013) 185–238, http://dx.doi.org/10.1007/s11831-013-9085-5.

[38] M. Beaudoin, H. Jasak, Development of a Generalized Grid Mesh Interface for Turbomachinery Simulations with OpenFOAM, 2008.

[39] U. Küttler, W.A. Wall, Comput. Mech. 43 (1) (2008) 61–72, http://dx.doi.org/10.1007/s00466-008-0255-5.

[40] B.M. Irons, R.C. Tuck, Internat. J. Numer. Methods Engrg. 1 (3) (1969) 275–277, http://dx.doi.org/10.1002/nme.1620010306.

[41] J. Degroote, K.-J. Bathe, J. Vierendeels, Comput. Struct. 87 (11–12) (2009) 793–801, http://dx.doi.org/10.1016/j.compstruc.2008.11.013.

[42] K.-J. Bathe, Finite Element Procedures, Prentice Hall, 1996, p. 1037.

[43] S. Turek, J. Hron, Fluid-Structure Interaction, Vol. 53, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 371–385, http://dx.doi.org/10.1007/3-540-34596-5_15.

[44] S. Turek, J. Hron, M. Razzaq, H. Wobker, M. Schäfer, in: H.-J. Bungartz, M. Mehl, M. Schäfer (Eds.), Lecture Notes in Computational Science and Engineering, in: Lecture Notes in Computational Science and Engineering, vol. 73, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 413–424, http://dx.doi.org/10.1007/978-3-642-14206-2, arXiv:1011.1669v3.

[45] X. Liao, L. Xiao, C. Yang, Y. Lu, Front. Comput. Sci. 8 (3) (2014) 345–356, http://dx.doi.org/10.1007/s11704-014-3501-3.

[46] ARCHER: The Latest Uk National Supercomputing Service.