Abstract

Concurrently coupled numerical simulations using heterogeneous solvers are powerful tools for modeling multiscale phenomena. However, major modifications to existing codes are often required to enable such simulations, posing significant difficulties in practice. In this paper we present a C++ library, i.e. the Multiscale Universal Interface (MUI), which is capable of facilitating the coupling effort for a wide range of multiscale simulations. The library adopts a header-only form with minimal external dependency and hence can be easily dropped into existing codes. A data sampler concept is introduced, combined with a hybrid dynamic/static typing mechanism, to create an easily customizable framework for solver-independent data interpretation. The library integrates MPI MPMD support and an asynchronous communication protocol to handle inter-solver information exchange irrespective of the solvers’ own MPI awareness. Template metaprogramming is heavily employed to simultaneously improve runtime performance and code flexibility. We validated the library by solving three different multiscale problems which also served to demonstrate the flexibility of the framework in handling heterogeneous models and solvers. In the first example, a Couette flow was simulated using two concurrently coupled Smoothed Particle Hydrodynamics (SPH) simulations of different spatial resolutions. In the second example, we coupled the deterministic SPH method with the stochastic Dissipative Particle Dynamics (DPD) method to study the the effect of surface grafting on the hydrodynamics properties on the surface. In the third example, we consider conjugate heat transfer between a solid domain and a fluid domain by coupling the particle-based energy-conserving DPD (eDPD) method with the Finite Element Method (FEM).

Keywords: multiscale modeling, concurrent coupling, multiphysics simulation, energy-conserving DPD, SPH, FEM, programming framework
a subset of a system that is handled by a single solver. The numerical modeling result on a domain may
depend on information from other subdomains of the system and vice versa.

An interface is a communication layer for exchanging information such as the boundary conditions
between two or more solvers. A sender is a solver process which is pushing information into an interface,
while a receiver is pulling information from the interface. A solver can be both sender and receiver at
the same time. Peers are the collection of receivers with regard to a given sender. A MUI interface is an
instance of our software interface enabling the physical interfaces.

2. Introduction

The potential of multiscale modeling lies in its ability of probing the properties of hierarchical systems
by capturing events that occur across a wide range of time and length scales that exceed the capability of
any single solver and method [1]. More recently, one specific branch, i.e. domain decomposition-based
concurrent coupling, has seen rapid development since it allows on-the-fly information exchange and
interaction between multiple simulation subdomains handled by different solvers.

Multiscale concurrent coupling using domain decomposition dates back to the classical Schwartz alter-
ning method [2, 3], where solutions of a partial differential equation (PDE) on two subdomains can be pur-
sued iteratively. In this method, the calculation for subdomain A is first constrained by a pseudo-boundary
embedded inside the other subdomain B, while the solutions on the pseudo-boundary are consistent with
the known/initial values of B. A similar calculation is then performed for subdomain B. This procedure is
repeated iteratively until convergence of solution in the hybrid region or global domain is achieved. In
general, there are two practical strategies to enforce the pseudo-boundaries of two subdomains. The first
is state-variable, e.g. density, velocity, etc., based coupling, where the constraints are placed on the state
variables on the two pseudo-boundaries alternatively. The other is flux based coupling, where the flux, e.g.
mass flux, momentum flux, etc., flowing into/out of one subdomain is compensated by the other subdomain
so that the laws of conservation are respected.

However, implementing concurrently coupled simulations remains difficult. On one hand, hard-coding
remains commonplace in projects that employ ad hoc approaches. Such practice can quickly become
an obstacle when further development is needed. On the other hand, despite the richness of available
coupling schemes and tools [4–9], adapting existing code to meet the programming interface specification
of a coupling framework frequently lead to code refactoring that consumes a substantial amount of man-
hours [10]. Such frameworks could also be awkward, especially for theorists without an expertise in
software engineering, when prototyping new coupling schemes.

As far as we know, a general and non-expert-friendly software library that assists the concurrent coupling
of independently developed solvers remains unavailable. The Multiscale Universal Interface (MUI) project
aims to fill in this gap by creating a light weight plugin library that can glue together essentially all
numerical methods including, but not limited to, Finite Difference, Finite Volume, Finite Element, Spectral
Method, Spectral Element Method, Lattice Boltzmann, Molecular Dynamics, Dissipative Particle Dynamics
and Smoothed Particle Hydrodynamics. Hence, it can deal with Lagrangian or Eulerian descriptions or a
mixture of both. It is expected to be able to accommodate a wide range of coupling schemes regardless of
the quantities being exchanged, the equations being solved, the time stepping pattern and/or the degree of
spatial and temporal separation.

In order to achieve such a high level of universality, MUI is designed to avoid defining the math
behind the coupling procedure, i.e. it does not specify which and how quantities are coupled. Instead,
it provides services to facilitate the effort of constructing arbitrary coupling schemes by enabling the
communication and interpretation of arbitrary physical quantities using arbitrary data types as demanded
by each participating solver.

MUI is simple to use, in the sense that existing solvers do not have to be refactored before using it.
MUI provides a very small set of programming interfaces instead of dictating any from the solver. The
entire library is coded in a header-only fashion with the Message Passing Interface (MPI) being the only
external dependency. Hence, it can be used in exactly the same way as the C++ standard library without
pre-compilation. It does not interfere with the intra-solver communication for solvers using MPI.
MUI is fast, in the sense that using it only consumes a small amount of CPU time as compared to that used by the solver itself. To achieve this goal we heavily employ the C++ generic programming/template metaprogramming feature to eliminate the abstraction overhead that may otherwise arise when maintaining the high-level flexibility of the framework.

3. Data Interpretation

3.1. Data Points

A universal coupling framework entails a generalized data representation framework. By observing the fact that discretization is the first step toward any numerical approximation, we realize that essentially every simulation system can be treated as a cloud of data points each carrying three attributes, i.e. position, type, and value, as shown in Fig. 1. The points might be arranged on a regular grid or connected by a certain topology in some of the methods, but for the sake of generality it is useful to ignore this information temporarily.

MUI defines a generic push method for solvers to exchange points carrying different types of data in a homogeneous fashion. The method assumes the signature:

```cpp
template<typename TYPE> inline bool push( std::string name, mui::point location, TYPE value );
```

The push method can accept data points of arbitrary type because it takes the type of the value as a template argument. Points belonging to the same physical variable, i.e. points pushed under the same
(a) Bilinear color interpolation

(b) Weighted average

Figure 2: Texture sampling vs. Gaussian kernel sampling.

name, are accumulated in a continuous container and sent out to receivers collectively to avoid fragmented communication. Note that the solver takes the responsibility of determining which points get pushed in, because the determination of the interface region uses mostly prior knowledge and hence it does not necessarily require direct aid from the coupling library.

3.2. Data Sampler

A generic fetch method for universal data interpretation on top of the data point representation is less trivial, however. The challenge of achieving universality here lies in the fact that solvers may be agnostic of the math and method used by their peers. Thus, a finite difference code might find itself in need of the value of pressure at grid point \((x_0, y_0)\), yet none of the vertices supplied by its peer finite element solver lies exactly on that point. In this case, the MUI interface is not supposed to simply throw out an exception. Possible solutions could be to use the pressure value defined at the nearest point, or to perform some sort of interpolation using nearby points. The decision for the best algorithm requires knowledge beyond the reach of MUI, but we implemented a flexible data interpretation engine so that users can choose to plug in an appropriate one with trivial effort.

Such engine is powered by the data sampler construct, which is derived from the concept of texture sampling in computer graphics \[11\]. A texture is essentially a rasterized image of discrete pixels being mapped onto some 3D surface. As a result of 3D projection and transformation, there is no one-to-one correspondence between the pixels of the surface as shown on the screen and the pixels on the texture, and the color of the texture at a fractional coordinate could be displayed. In this case, as shown in Fig. 2a, the graphics hardware performs an interpolation (usually bilinear) of the pixel values adjacent to the requested fractional coordinate, and return the interpolated value as the color at the requested point.

The MUI data sampler works in a similar, but enhanced, way: each physical quantity is treated as a samplable object, while data samplers are used to interpolate values from the cloud of discrete data points contained in it. A sampler is a class implementing the interfaces filter and support as shown by the example of the MUI built-in Gaussian kernel sampler in Listing 1. A line-by-line explanation of the C++ code is given below:

- line 1: class template argument declaration. By making the input and output type of samplers template arguments, it is possible to reuse the same interpolating algorithm without duplicating the code merely for the different data types used in different solvers.
- line 4-6: The internal basic data types of MUI are globally parameterized in a configuration class as detailed in Sec.5.
- line 8-11: Constructor that sets up the shape parameters of the Gaussian kernel.
• line 13-27: The filter method performs data interpolation/interpretation using data points fed by MUI. The MUI virtual container object maps to the subset of the data points that falls within the sampler’s support while its usage pattern resembles that of std::vector.

• line 29-31: MUI uses geometry information provided by the support method as the extent of the sampler’s support to efficiently screened off outlying particles with an automatically tuned spatial searching algorithm.

• line 33-36: storage for sampler parameters, etc.

Listing 1: The implementation of the MUI built-in Gaussian kernel sampler

```cpp
template<typename OTYPE, typename ITYPE=OTYPE, typename CONFIG=default_config>
class sampler_gauss {
public:
    using REAL = typename CONFIG::REAL;
    using INT = typename CONFIG::INT;
    using point_type = typename CONFIG::point_type;

    sampler_gauss( REAL r_, REAL h_ ) :
        r(r_),
        h(h_),
        nh(std::pow(2*PI*h,-0.5*CONFIG::D)) {}

    template<template<typename,typename> class CONTAINER>
    OTYPE filter( point_type focus, const CONTAINER<ITYPE,CONFIG> &data_points ) const {
        REAL wsum = 0;
        OTYPE vsum = 0;
        for(INT i = 0 ; i < data_points.size() ; i++) {
            auto d = (focus-data_points[i].first).normsq();
            if ( d < r*r ) {
                REAL w = nh * std::exp( (-0.5/h) * d );
                vsum += data_points[i].second * w;
                wsum += w;
            }
        }
        if ( wsum ) return vsum / wsum;
        else return OTYPE(0);
    }

    geometry::any_shape<CONFIG> support( point_type focus ) const {
        return geometry::sphere<CONFIG>( focus, r );
    }

protected:
    REAL r;
    REAL h;
    REAL nh;
};
```

The sampling procedure works as:

1. Solver invokes the fetch method of MUI with a point of interest and a sampler;
2. MUI collects all points that lies within the sampler’s support around the point of interest into a virtual container;
3. MUI feeds the sampler with the collected points and lets the sampler perform its own interpolation;
4. The sampler returns the interpolation result back to the user/solver through MUI.

MUI achieves generality in interpolation by allowing users to easily create new samplers to express custom approximation algorithm that can leverage domain-specific knowledge of the system. The value at an arbitrary desired location can be obtained by using samplers that interpolate values from nearby points. In addition, a single piece of sampler code can be used for different data types, e.g. float, double or int, because the filter and fetch method take the type of the data points as a template argument.

The sampling framework makes it possible for users to fully focus on the design of algorithms while delegating the data management job to MUI. To further simplify the usage, MUI includes several predefined
samplers such as a Gaussian kernel sampler, a nearest neighbor sampler, a moving average sampler, an exact point sampler, etc.

3.3. Typing system

MUI implements a hybrid dynamic/static typing system to combine the performance of static typing with the flexibility of dynamic typing. The dynamic typing behavior of MUI is controlled at the level of physical quantities. The first value of the data point received by the push method for each name determines the type of the physical quantity, while the type of subsequently pushed data points will be examined against the type of the existing storage object. The entire storage object is type-dispatched only once on the receiver’s side for each sampling request. Hence, MUI does not have to perform the expensive type-dispatching for each data points. This coarse-grained dynamic-typing technique is especially important for sampling where data points are being frequently accessed.

The system uses a type list to enumerate all possible types that may be handled by MUI and to automatically generate type-dispatching code. A type list is essentially an instantiation of the variadic class template mui::tuple with the template arguments being the list members. Using recursive templates we can either query the type of a list member using an index (a compile-time constant) or check the index of a type in a given list. A default type list containing frequently used C++ built-in data types is predefined in MUI's default configuration.

Support for new types can be trivially added into MUI by 1) adding the type into the predefined type list; and 2) defining the insertion and extraction operator of the type with regard to mui::istream and mui::ostream. mui::istream and mui::ostream are the data serialization classes in MUI sharing the same usage pattern with std::iostream. Since MUI predefines the insertion and extraction operators for all C++ primitive types, an overloaded operator for any composite type can be implemented easily in terms of the primitive ones as illustrated in Listing 2.

Listing 2: The insertion and extraction operator for type bond can be overloaded in a straightforward way as shown below.

```cpp
// bond is a composite data type
struct bond {
    // double is a primitive data type
    double k, r;
};

// overload serialization operator
define the insertion and extraction operator of the type with regard to mui::istream and mui::ostream are the data serialization classes in MUI sharing the same usage pattern with std::iostream. Since MUI predefines the insertion and extraction operators for all C++ primitive types, an overloaded operator for any composite type can be implemented easily in terms of the primitive ones as illustrated in Listing 2.

Listing 2: The insertion and extraction operator for type bond can be overloaded in a straightforward way as shown below.

```cpp
1 // bond is a composite data type
2 struct bond {
3     // double is a primitive data type
4     double k, r;
5 };
6
7 // overload serialization operator
8 void mui::ostream& operator <<( mui::ostream& ost, const bond& v ) {
9     // enumerate over primitive members
10     return ost << v.k << v.r;
11 }
12
13 // overload deserialization operator
14 void mui::istream& operator >> ( mui::istream& ist, bond& v ) {
15     return ist >> v.k >> v.r;
16 }
```

3.4. Storage and Time coherence

Regardless of the actual simulation algorithm, the main body of a solver is essentially a time marching loop in which the quantity of interest is being iteratively solved. Hence, points of the same quantity may be sent to a MUI interface repeatedly during a simulation. However, it is inevitable that one solver may run faster than its peer due to factors such as intrinsic performance disparity, load imbalance and transient interruption. In such situation, data points from a later time step may override previous ones belonging to the same quantity before the receiver could ever get a change to sample them.

To address this problem, MUI stores the collection of numerical results generated during each time step as a frame. Frames are indexed by their timestamps so different frames do not override each other. Technically, all data points being pushed in for a single physical quantity within a single time step are collectively stored in an instance of mui::any_storage, MUI's dynamically typed data container. A frame is essentially a std::map between the quantity names and the actual mui::any_storage instances, while the
frames themselves are again organized in a `std::map` using the time stamp as the key. This sparse storage structure, as illustrated in Figure 3, allows efficient allocation of memory regardless of whether the time frames are equally distributed or not. It also allows each physical quantity to be selectively committed in a subset of all time frames.

The memory allocation for frames is managed transparently by a buffering scheme. The deallocation, however, must be set up by the user because it is impossible to predict whether a frame will be reused in the future. Utility methods are provided for the user to either explicitly request the disposal of time frames or to let MUI automatically discard frames that are older than a certain age in the simulation units. The default memory length is infinity so no frames will be freed automatically.

In situations where batches of data points have to be moved between components of MUI, we use the C++11 `std::move` semantic to avoid duplicate memory allocation and copy.

A set of `time samplers` are also predefined in MUI. Time samplers work in essentially the same way as the data samplers, except for that they are one-dimensional along the time axis and use the output from a spatial sampler as the input. In Figure 4 we demonstrate the concept of a simple averaging sampler. It is also straightforward to implement more sophisticated time samplers with features such as filtering or prediction.

4. Parallel Communication

4.1. MPI Multiple-Program-Multiple-Data Setup

MUI uses MPI as the primary communication mechanism due to its portability, ubiquity, efficiency and compatibility with existing codes. In addition, the multiple-program-multiple-data (MPMD) mode is a natural fit for the purpose of concurrent coupling because it allows the users to launch multiple ranks for each of the solvers involved.

To ensure that the MPMD topology is only visible to MUI itself and hidden from the solver code, it is mandated by MUI that the solver should make no direct reference to the MPI predefined constant `MPI_COMM_WORLD` for any of its own communications. Instead, a globally accessible variable of type `MPI_Comm` should be defined to hold the global communicator, which can be obtained from a MUI helper function call.
Figure 4: Time samplers can interpolate or extrapolate the output of a spatial sampler over a range of time frames.

that effectively splits MPI_COMM_WORLD into subdomains using the MPI appnum variable. This is in fact one of the few modifications to the solvers that is ever dictated by the MUI.

In order to identify and connect MPI ranks belonging to the different domains, a URI string must be used to initialize MUI with the format mpi://domain/interface. The protocol field, e.g. mpi, is used for dynamic allocation of the correct communication class through the object factory mechanism. This would allow new communication implementations to be added in a straightforward manner. As shown in Algorithm 1, the domain field is used to split the world domain, while the interface field is used to match up peer ranks to form the intercommunicator across solvers.

**Algorithm 1** MPI MPMD setup.

```plaintext
function InitMpiMpmd(URI)
    // Duplicate world communicator to avoid interfere with solver
    World ← MPI_COMM_DUP(MPI_COMM_WORLD)
    GlobalSize ← MPI_COMM_SIZE(World)
    // Parse URI string
    DomainString, InterfaceString ← Parse(URI)
    DomHash ← std::Hash(DomainString)
    IfsHash ← std::Hash(InterfaceString)
    // Create local and inter-communicators for solver
    LocalDomain ← MPI_COMM_SPLIT(World, DomHash)
    AllDomHash[0..GlobalSize-1] ← MPI_ALLGATHER(DomHash)
    AllIfsHash[0..GlobalSize-1] ← MPI_ALLGATHER(IfsHash)
    root ← min i | AllDomHash[i] ≤ DomHash & AllIfsHash[i] = IfsHash
    RemoteDomain ← MPI_INTERCOMM_CREATE(LocalDomain, 0, World, root, IfsHash)

    return LocalDomain, RemoteDomain
end function
```

8
Figure 5: An example illustrating the effect of smart sending. The green domain is handled by solver A and spatially decomposed between 4 MPI ranks, while the blue domain is handled by solver B using 3 MPI ranks. By checking for the overlap between the interface regions owned by the ranks, MUI can eliminate unnecessary data transfer between ranks such as $A_0 - B_1$, $A_2 - B_2$, and $A_3 - B_0$ etc.

4.2. Asynchronous I/O and smart sending

MUI assumes an asynchronous communication model because it can be difficult to find synchronization points between multiple heterogeneous solvers. Specifically, MPI collective methods are not used. Instead, MUI makes use of point-to-point non-blocking send and blocking receive methods. The send buffers are stored in a queue alongside with their corresponding MPI requests, and are freed upon completion of the communication. Whenever MUI finds itself in need of data (e.g., due to a fetch request), it continuously accepts incoming MPI messages while also testing for the completion of pending sends until the arrival of the needed data. This asynchronicity is encapsulated within MUI and is completely transparent to the solver, whereas a non-blocking test method is also provided for advanced users to query the availability of data.

An optional smart sending feature is also introduced to optimize the amount of MPI messages. It is a selective communication mechanism based on spatial overlap detection. Each solver instance can define two regions of interest, i.e. a fetch region and a push region, through a Boolean combination of geometric primitives such as spheres, cuboids and points. As illustrated in Figure 5, the regions are broadcasted among all the processes so that the communication between a sender and a receiver whose region of push/fetch has no overlap can be safely eliminated. In this way, the communication made by each MUI instance can be localized to a few peers who are really in need of the data. To accommodate the case of moving boundaries, each region of interest is associated with a validity period. The smart sending feature can be safely ignored for convenience because both regions would default to $(-\infty, +\infty)$ with a validity period of infinity as a safety fall-back.

5. Customizability

In addition to the vast customization space regarding communication content, spatial and temporal interpolation algorithm and communication pattern, MUI allows a number of its low-level traits to be parameterized at compile-time using a configuration class. A default configuration class \texttt{default_config} is shown in Listing 3. The class serves as the last template argument of all MUI component classes and samplers. The class is passed along during inheritance and member definition so users only need to specify it once when instantiating the MUI top-level object. It allows the tweaking of:
Figure 6: MUI automates the communication of point data across solvers. Flexibility of interpolation is achieved by allowing the expression of interpolation algorithms as samplers, which are used by the fetch method.

- dimensionality of the physical space;
- precision of floating point numbers;
- integer width;
- time stamp type;
- type list (as mentioned in Section 3.3);
- debugging switch;
- exception handling.

Such static configuration mechanism can eliminate unnecessary runtime polymorphic overhead and also allows MUI to receive better performance optimization during the compilation phase.

Listing 3: The default configuration class for MUI

```c
struct default_config {
    static const int D = 3;
    using REAL = double;
    using INT = int64_t;
    using point_type = point<REAL,D>;
    using time_type = REAL;
    static const bool DEBUG = false;

    template<typename... Args> struct type_list_t {}
    using type_list = type_list_t<int,double,float>;
    using EXCEPTION = exception_segv;
};
```

6. Demonstration Examples

6.1. Couette flow: SPH-SPH coupling

MUI assumes a push-fetch workflow and serves as the data exchange and interpretation layer between solvers as visualized in Figure 6. To demonstrate the real-world usage pattern of MUI, we present a minimal-working-example (MWE) benchmark of concurrently coupled Smoothed Particle Hydrodynamics (SPH) simulation.

The algorithm is based on a velocity coupling scheme described in Algorithm 2. As illustrated in Figure 7, the system of interest was simulated using two overlapping SPH domains, i.e. a lower one...
The flow was simulated by two separate SPH domains. Each domain contains a send region and a receive region that are not overlapping with each other.

and an upper one, using either same or different resolutions. During each time step, the velocity of the SPH particles lying within the receiving part of the overlapped region is set as the average velocity of nearby particles from the other domain as interpolated using a SPH quintic interpolation sampler. We used LAMMPS [12] as the baseline solver, and inserted only about 70 lines of code to implement the algorithm using MUI as given in Listing 1 in Support Information (SI).

Algorithm 2 SPH-SPH coupling scheme. The C++ code for the Quintic and ExactTime samplers are given in Listing 2 and Listing 3 in SI, respectively.

We then used the MUI-enabled LAMMPS to model a Couette flow by solving the Navier-Stokes equation. A system of a unit cube was simulated. The volumetric number densities of SPH particles were $20^3$ and $40^3$ for the lower and upper domains, respectively. As shown in Figure 8 the velocity profile obtained from the coupled simulation is consistent with the analytic solution.

The SPH simulation was performed on a workstation with two hexa-core Intel Xeon E5-2630L CPUs running at 2.0GHz. Table 1 shows a breakdown list of total CPU time spent in different parts of the LAMMPS SPH solver in the aforementioned SPH/SPH simulation. MUI consumes less than 5% of the total CPU time. Note that this includes both the sampling computation time and the communication and
Figure 8: Transient velocity profile of a Couette flow. The system was modeled by two SPH domains of different resolutions. Numerical solutions obtained from the lower and upper domains are plotted in green and blue, respectively. The analytic solution is shown by dashed line, while the interface region is indicated by the shaded box.

Table 1: Couette flow: a breakdown of the CPU time usage.

| Procedure   | MPI 6x6 |
|-------------|---------|
| Overall     | 400.1   |
| MUI         | 16.27   |
| push        | 0.24    |
| fetch       | 8.81    |
| sampler     | 6.73    |
| point query | 1.63    |
| communication | 6.77   |
Table 2: Soft matter: parameters for the SPH-DPD simulation

| arg   | val       | arg   | val       |
|-------|-----------|-------|-----------|
| $N_p$ | 84,375    | $N_p$ | 4,000     |
| $\rho$ | 5         | $\rho_N$ | 0.064     |
| $\gamma$ | 4.5       | $\rho_m$ | 5         |
| $\sigma$ | 3.0       | $\eta$ | 3.72      |
| $\delta t$ | 0.01     | $\delta t$ | 0.5     |
| $r_c$ | 1.0       | $r_c$ | 10        |
| $k_BT$ | 1.0       | $c_s$ | 1.5       |
| $g$ | 0.001     | $g$ | 0.001     |
| $w_D = w_0^{0.25}$ |           |       |           |

Table 3: Repulsive force constants $a_{ij}$ for DPD

|          | wall | dendrimer | solvent |
|----------|------|-----------|---------|
| wall     | 15   | 15        | 15      |
| dendrimer| 15   | 15        | 75      |
| solvent  | 15   | 75        | 15      |

bookkeeping overhead, while the sampling time is actually part of the useful work and hence should not be counted as overhead.

6.2. Soft Matter: SPH-DPD coupling

Next we demonstrate a concurrently coupled deterministic/stochastic simulation using a similar coupling scheme. As illustrated in Figure 9, the flow between two parallel infinitely-large plates driven by a uniform body force was simulated. The upper plate corresponds to a simple no-slip boundary, while the lower plate is grafted by a hydrophobic fourth order binary dendrimers. We used the coupled simulation to investigate the effect of coating on the hydrodynamics of the system.

All quantities/parameters mentioned in this simulation are in reduced DPD units. A system of size $40 \times 110 \times 40$ is constructed using an upper domain and a lower domain. The flow field in the upper domain, which has a simpler boundary condition, is simulated using the Smoothed Particle Hydrodynamics (SPH) method solving the Navier-Stokes equation. The flow field in the lower domain is simulated using Dissipative Particle Dynamics (DPD) solving Newton’s equation of motion in stochastic form. The DPD domain spans from $y = -1$ to $y = 28$, and includes the solvent and a stationary wall lying between $y = -1$ to $y = 0$ with its upper surface grafted by 160 fourth order binary dendrimers. The surface coverage of the dendrimers is 70%. The viscosity of the DPD solvent is measured as 3.72. The SPH domain spans from $y = 20$ to $y = 110$ and included a stationary upper wall lying between $y = 100$ to 110 which serves to enforce the no-slip boundary condition. A gravity of 0.001 in the x direction is imposed for both domains. The DPD simulation was carried out on 4 nVidia GeForce GTX TITAN GPUs using the *USER Meso* package [13]. The parameters for setting up the simulation are listed in Table 2.

The coupling scheme is similar to that used in the previous SPH-SPH simulation as described in Section 6.1. However, the SPH solver assumes a time step size which is 50 times that of the DPD solver. Accordingly, as demonstrated in Algorithm 3, the SPH domain samples the average velocity of the DPD domain over the last 50 frames to smooth out the randomness, while the DPD domain always samples the latest time frame sent from the SPH domain. The velocity profile converged to that of a Poiseuille flow after 10000 DPD units. As shown in Figure 10 the effective channel width is reduced by the hydrophobic coating by about 0.5 DPD unit.
Algorithm 3 SPH-DPD coupling scheme. The C++ code for the SumOver sampler is given in Listing 4 in SI.
Figure 9: A hybrid system consisting of an SPH upper domain and a DPD lower domain was modeled to study the hydrodynamical properties of dendrimer grafted surface. DPD wall particles, dendrimers, DPD solvent particles and SPH fluid particles are rendered in black, green, red and blue, respectively.

6.3. Conjugate Heat Transfer

We further demonstrate a coupled Eulerian/Lagrangian simulation of the cooling process of a heating cylinder immersed in a channel flow using the energy-conserving Dissipative Particle Dynamics (eDPD) method [14] and the finite element method (FEM). The eDPD model is an extension to the classical DPD model [15, 16] with explicit temperature and heat transferring terms. The FEM solver can solve the time-dependent heat equation

$$\frac{\partial T}{\partial t} = -\alpha \Delta T + f$$

where $\alpha$ is the thermal diffusivity and $f = f_0 + f_{\text{interf}}$, the heat source.

The coupling scheme is shown in Algorithm 4. The FEM domain pushes the temperature of boundary vertices, with which the eDPD solver calculates the heat flux generated by each particles surrounding the cylinder and pushes them as data points. The FEM solver averages the fluxes computed by eDPD and assigns the result to boundary vertices based on a Voronoi diagram of the vertices. The heat flux value is used as the Neumann boundary condition required in solving the time-dependent Poisson equation. The accuracy of the scheme was validated by solving for the temperature profile in a quartz-water-quartz system whose left and right boundary were fixed at 270K and 360K as shown in Figure 11. The thermal diffusivities of water and quartz were assumed to be constant at $0.143 \times 10^6 m^2/s$ [17] and $1.4 \times 10^6 m^2/s$ [18].
The velocity profile derived from the coupled SPH/DPD simulation converges to that of a Poiseuille flow. SPH results, DPD results and the analytic solutions are plotted in blue dots, green dots and dashed line, respectively. The background picture indicates the system composition at the corresponding y position.

over the temperate range, respectively, while the interfacial thermal diffusivity was chosen as the arithmetic mean between the two values.

A system composed of a 3D fluid domain filled with water, a 2D solid domain and a fluid-solid interface was then simulated using the validated scheme. The entire domain is periodic in x and z direction, but is bounded by a pair of no-slip infinite walls of constant temperature in the y direction. The fluid domain and the walls are simulated using eDPD, while the solid domain is solved using FEM. Parameters and scaling factors used to set up the eDPD and FEM calculations are given in Table 4. The simulation result is shown in Figure.12. The Reynolds number is defined by \( Re = \frac{v_{\text{max}} D}{\nu} \) where \( v_{\text{max}} = 0.65 \frac{L_0}{\tau} \) is the maximum inlet velocity, \( \nu = 6.62 \) the kinematic viscosity and \( D = 20.0 L_0 \) the diameter of the cylinder.

We obtained a smooth temperature transition in the hybrid domain using MUI and the aforementioned coupling scheme. The example demonstrates the capability of MUI in coupling two different fields, e.g. a flow field and a thermal field. The method can facilitate the study of inhomogeneous coolants, e.g. colloidal suspensions, thanks to the flexibility brought about by the particle method.

7. Related work

Many software tools and frameworks have been proposed for carrying out concurrently coupled multiscale simulation. However, MUI differentiates itself from the existing ones by its ease of use, universaility, code reusability and out-of-the-box parallel communication capability.

The majority of existing frameworks focus on the coupling between solvers that deal with PDEs using grid/mesh-based methods. The Core Component Architecture (CCA) specification [4–6] appears to be the most widely adopted standard for this types of coupling with conforming implementations such as Uintah [7–9], CCAT [19], SCIRun2 [20] and so on [21, 22]. The CCA framework invokes individual solvers as components at runtime and let components register interface functions, called ports, that are to be called by other components for information exchange. Different from MUI, the CCA specification does not specify the input and output arguments of the ports, and it is up to the solvers to determine the content of communication. The CCA core specification also does not define a mechanism for inter-solver communication outside of the shared memory space of a single process. There exist also coupling frameworks that require more extensive modification of existing solvers. For example, solving algorithms are encapsulated as classes that
Algorithm 4 eDPD-FEM coupling scheme. The C++ code for the VoronoiMean and Linear samplers are given in Listing 5 and Listing 6 in SI, respectively.

```cpp
for t = 0; t < T\text{Total}; do
  > eDPD domain
  for i = 1; i < N\text{Particles}; do
    if WithinCutoffOfCylinder(i) then
      S\text{spatial} ← Linear(h_{\text{max}})
      S\text{temporal} ← ExactTime(\varepsilon)
      T\text{wall} ← MUI::Fetch("T", coord[i], t_{\text{FEM}}, S\text{spatial}, S\text{temporal})
      q ← PerParticleHeatFlux(T[i], T\text{wall})
      MUI::Push("q", coord[i], \frac{q}{C_v})
    end if
  end for
  MUI::Commit(t)
  if Mod(t, 10\delta t) = 0 then
    MUI::Forget(t - 10\delta t)
  end if
end for

> FEM domain
for t = 0; t < T\text{Total}; do
  > Push
  for each boundary vertex i do
    MUI::Push("T", coord[i], T[i])
  end for
  MUI::Commit(t)
  > Fetch
  for each boundary vertex i do
    S\text{spatial} ← VoronoiMean(Vertices)
    S\text{temporal} ← MeanOver(10\delta t)
    \eta_{\text{Interface}}[i] ← MUI::Fetch("\eta", coord[i], 1, S\text{spatial}, S\text{temporal})
  end for
  MUI::Forget(t)
  SolveForNextStep
end for
```

Figure 11: Heat conduction: temperature profile obtained for a quartz-water-quartz tri-layer system with the FEM solver handling the solid domain and the eDPD solver handling the fluid domain.
Table 4: Conjugate heat transfer: parameters for the eDPD-FEM simulation of immersed heating cylinder are taken from Ref [14].

| Parameter       | eDPD | FEM  |
|-----------------|------|------|
| length scale    | $L_0 = 11\text{nm}$ |     |
| time scale      | $\tau = 0.935\text{ns}$ |     |
| temperature scale | $T_0 = 300\text{K}$ |     |
| mass scale      | $m_0 = 3.32 \times 10^{-22}\text{kg}$ |     |
| $\alpha$        | $1.43 \times 10^{-7}\text{m}^2/\text{s}$ | $1.43 \times 10^{-7}\text{m}^2/\text{s}$ |
| $\delta t$      | $0.0125\tau$ | $0.125\tau$ |
| $\nu$           | $8.57 \times 10^{-7}\text{m}^2/\text{s}$ |     |

Figure 12: Conjugate heat transfer: the hybrid particle/mesh structure of the domain is shown by the upper plot. Streamlines and temperature distribution in the system are visualized in the lower plot by white lines and color-mapped contours, respectively.
expose a given set of interfaces in the Kratos [23] framework. The DDEMA [24] framework focuses on PDE solvers using mesh-based methods and casts software packages into an actor design pattern.

In addition, various specialized coupling frameworks have been proposed. The Macro-Micro-Coupling [25, 26] framework can solve coupling problems between macroscopic models and microscopic models. The MCI [27] framework is able to couple massively parallel spectral-element simulations and particle-based simulations in a highly efficient and scalable manner. The triple-decker algorithm can solve multiscale flow fields by coupling the Molecular Dynamics method, the Dissipative Particle Dynamics method and the incompressible Navier-Stokes equations [28]. The MUPHY [29] framework couples Lattice-Boltzmann method with molecular dynamics simulation. The MUSE [30, 31] framework is specialized in astrophysics simulation. In contrast, MUI facilitates the construction of a plug-and-play pool of any combination of particle-based and continuum-based solvers for solving multiscale problems without code rewriting. To the best of our knowledge, there is currently no other project that can achieve such level of generality.

8. Conclusion

In this paper we presented the Multiscale Universal Interface library as a generalized approach of coupling heterogeneous solver codes to perform multi-physics and multiscale simulations. The library assumes a solver/scheme-agnostic approach in order to accommodate as many numerical methods and coupling schemes as possible, while still maintains a simple and straightforward programming interface. The data sampler concept is the key enabling technique for this flexible framework of data interpretation. The library employs techniques such as dynamic typing, MPI MPMD execution, asynchronous I/O, generic programming and template metaprogramming to improve both performance and flexibility. Benchmarks demonstrate that the library can be adopted easily for coupling heterogeneous simulations.

Acknowledgment

This work was supported by the Department of Energy (DoE) Collaboratory on Mathematics for Mesoscopic Modeling of Materials (CM4) and the Air Force Office of Scientific Research (FA9550-12-1-0463). Particle simulation results were visualized with VMD [32, 33].

Reference

[1] E. Weinan. Principles of multiscale modeling. Cambridge University Press, 2011.
[2] P.-L. Lions. On the schwarz alternating method. i. In First international symposium on domain decomposition methods for partial differential equations, pages 1–42. Paris, France, 1988.
[3] J. Xu. Iterative methods by space decomposition and subspace correction. SIAM review, 34(4):581–613, 1992.
[4] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl. The CCA core specification in a distributed memory SPMD framework. Concurrency and Computation: Practice and Experience, 14(5):323–345, 2002.
[5] S. Lefantzi, J. Ray, and H. Najm. Using the common component architecture to design high performance scientific simulation codes. In Parallel and Distributed Processing Symposium, 2003. Proceedings. International, pages 10 pp.–, April 2003.
[6] L. McNes, B. Allan, R. Armstrong, S. Benson, D. Bernholdt, T. Dahlgren, L. Diachin, M. Krishnan, J. Kohl, J. Larson, S. Lefantzi, J. Nieplocha, B. Norris, S. Parker, J. Ray, and S. Zhou. Parallel PDE-based simulations using the common component architecture. In A. Bruaset and A. Tveito, editors, Numerical Solution of Partial Differential Equations on Parallel Computers, volume 51 of Lecture Notes in Computational Science and Engineering, pages 327–381. Springer Berlin Heidelberg, 2006.
[7] J. Davison de St.Germain, J. McCorquodale, S. Parker, and C. Johnson. Uintah: a massively parallel problem solving environment. In High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on, pages 33–41, 2000.
[8] S. G. Parker. A component-based architecture for parallel multi-physics PDE simulation. Future Generation Computer Systems, 22(1–2):204 – 216, 2006.
[9] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C. A. Wight, and J. R. Peterson. Uintah: A scalable framework for hazard analysis. In Proceedings of the 2010 TeraGrid Conference, TG ’10, pages 3:1–3:8, New York, NY, USA, 2010. ACM.
[10] B. Allan, S. Lefantzi, and J. Ray. ODEPACK++: refactoring the Isode fortran library for use in the cca high performance component software architecture. In High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on, pages 109–119, April 2004.
[11] J. F. Hughes, S. K. Feiner, J. D. Foley, K. Akeley, M. McGuire, A. v. Dam, and D. F. Sklar. Computer graphics: principles and practice. 2013.
