Journal of Computational Physics 297 (2015) 13–31

Contents lists available at ScienceDirect

Journal of Computational Physics

www.elsevier.com/locate/jcp

Multiscale Universal Interface: A concurrent framework for coupling heterogeneous solvers [☆], [☆☆]

Yu-Hang Tang [a], Shuhei Kudo [b], Xin Bian [a], Zhen Li [a], George Em Karniadakis [a,c,*]

[a] Division of Applied Mathematics, Brown University, Providence, RI, USA
[b] Graduate School of System Informatics, Kobe University, 1-1 Rokkodai-cho, Nada-ku, Kobe, 657-8501, Japan
[c] Collaboratory on Mathematics for Mesoscopic Modeling of Materials, Pacific Northwest National Laboratory, Richland, WA 99354, USA

A R T I C L E   I N F O

Article history:
Received 23 October 2014
Received in revised form 7 March 2015
Accepted 2 May 2015
Available online 14 May 2015

Keywords:
Multiscale modeling
Concurrent coupling
Multiphysics simulation
Energy-conserving DPD
SPH
FEM
Programming framework

A B S T R A C T

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 serve 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 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).

[☆] The authors wish it to be known that the first two authors each contributed indispensably in this interdisciplinary work and thus, in their opinion, should be regarded as co-first authors.
[☆☆] The Multiscale Universal Interface library is freely available under the GPLv3 license at http://www.cfm.brown.edu/repo/release/MUI/.
[*] Corresponding author at: Division of Applied Mathematics, Brown University, Providence, RI, USA.
E-mail addresses: yuhang_tang@brown.edu (Y.-H. Tang), shuhei-kudo@outlook.jp (S. Kudo), xin_bian@brown.edu (X. Bian), zhen_li@brown.edu (Z. Li), george_karniadakis@brown.edu (G.E. Karniadakis).

http://dx.doi.org/10.1016/j.jcp.2015.05.004
0021-9991/© 2015 Elsevier Inc. All rights reserved.

1. Glossary

A **solver** is a computer program that can carry out a given type of numerical simulation. It may execute in the single-program-multiple-data (SPMD) mode for parallelization. The same solver can be invoked multiple times separately during a certain simulation.

A **simulation** is the act of using one or multiple solvers to perform a numerical modeling task. A **system** is the entire set of physical time–space involved in a simulation. A **subdomain**, or simply **domain**, is 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 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 [1–3]. 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 alternating method [4,5], where solutions of a partial differential equation (PDE) on two subdomains can be pursued iteratively. In this method, the calculation for subdomain A is first performed with an enforced pseudo-boundary which extends into the other subdomain B, while the solutions on the pseudo-boundary are dictated to be the known values of B at corresponding locations. 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-boundary between 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 alternately [6,7]. 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 [8,9].

Despite existing theoretical developments, implementing concurrently coupled simulations remains difficult. On one hand, hard-coding remains commonplace in projects that employ *ad hoc* coupling 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 [10–15], adapting existing code to meet the programming interface specification of a coupling framework frequently leads to code refactoring that consumes a substantial amount of man-hours [16]. Such frameworks could also be cumbersome, 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 Method, 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 existing intra-solver communications for solvers using MPI.

MUI is also 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.

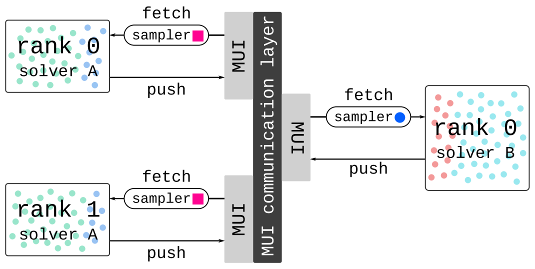

Fig. 1. MUI facilitates the exchange of information across solvers by letting solvers push and fetch data points. Flexibility of interpolation is achieved by allowing the expression of interpolation algorithms as samplers as well as by accepting data points of arbitrary types.

3. Data interpretation framework

3.1. Usage overview

As visualized in Fig. 1, MUI assumes a **push–fetch** workflow and serves as the data exchange and interpretation layer between solvers. While more concrete examples on the usage of MUI are given in Section 6, the list below outlines the typical steps for incorporating MUI into an existing solver:

1. Substituting the MPI global communicator (Section 4.1);
2. Allocating MUI objects;
3. Identifying code regions that supply information to peer solvers and pushing the data as points using MUI (Section 3.2);
4. Identifying code regions that require information from peer solvers and fetching through MUI's sampling interface (Section 3.3);
5. Configuring inter-solver synchronization (Section 3.5);
6. Optimizing performance by managing memory allocation, simplifying communication topology and tuning low-level traits etc. (Section 3.5, Section 4.2, and Section 5). This step is not mandatory for obtaining correct results but may have an impact on simulation efficiency.

3.2. 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. 2. 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 unified fashion. The method assumes the signature:

```
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 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.

3.3. 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 weighted interpolation using nearby points as shown in Fig. 3(a). 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 minimal effort.

Such engine is enabled by the *data sampler* construct, which is derived from the concept of texture sampling in computer graphics [17]. A texture is essentially a rasterized image of discrete pixels being mapped onto some 3D surface. As a result

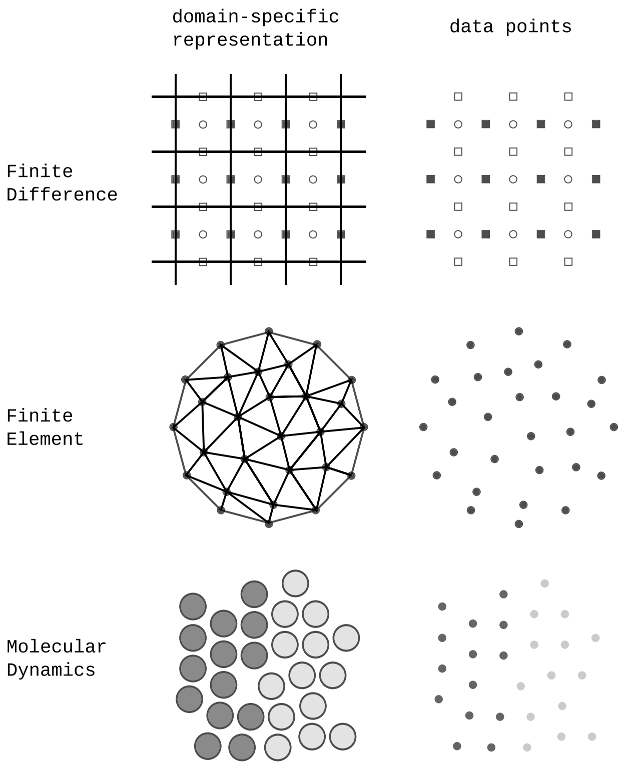

Fig. 2. Any discrete systems can be generalized as a cloud of data points by ignoring the domain-specific knowledge such as meshes.

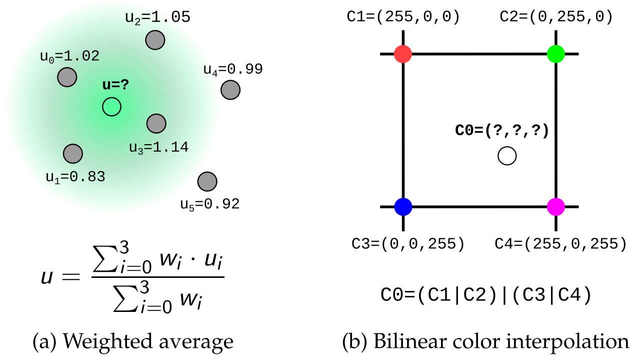

Fig. 3. Weighted kernel sampling and texture sampling. (For interpretation of the colors in this figure, the reader is referred to the web version of this article.)

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. 3(b), 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 MUI 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.
- lines 4–6: The internal basic data types of MUI are globally parameterized in a configuration class as detailed in Section 5.
- lines 8–11: Constructor that sets up the shape parameters of the Gaussian kernel.

- lines 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**.
- lines 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.
- lines 33–36: storage for sampler parameters, etc.

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

```
 1 template<typename OTYPE, typename ITYPE=OTYPE, typename CONFIG=default_config>
 2 class sampler_gauss {
 3 public:
 4   using REAL = typename CONFIG::REAL;
 5   using INT = typename CONFIG::INT;
 6   using point_type = typename CONFIG::point_type;
 7
 8   sampler_gauss( REAL r_, REAL h_ ) :
 9     r(r_),
10     h(h_),
11     nh(std::pow(2*PI*h,-0.5*CONFIG::D)) {}
12
13   template<template<typename,typename> class CONTAINER>
14   OTYPE filter( point_type focus, const CONTAINER<ITYPE,CONFIG>&data_points ) const {
15     REAL wsum = 0;
16     OTYPE vsum = 0;
17     for(INT i = 0 ; i < data_points.size() ; i++) {
18       auto d = (focus-data_points[i].first).normsq();
19       if ( d < r*r ) {
20         REAL w = nh * std::exp( (-0.5/h) * d );
21         vsum += data_points[i].second * w;
22         wsum += w;
23       }
24     }
25     if ( wsum ) return vsum / wsum;
26     else return OTYPE(0);
27   }
28
29   geometry::any_shape<CONFIG> support( point_type focus ) const {
30     return geometry::sphere<CONFIG>( focus, r );
31   }
32
33 protected:
34   REAL r;
35   REAL h;
36   REAL nh;
37 };
```

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 lie 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** methods 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.

3.4. 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 performed at the level of physical quantities. The value of the first data point received by the **push** method for each physical quantity determines the type of the 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 a variadic class template 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 the data serialization classes in MUI which share the same usage pattern with the C++ standard input and output streams. 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. **`mui::istream`** and **`mui::ostream`** are the built-in data serialization classes in MUI.

```
// bond is a composite data type
struct bond {
  // double is a primitive data type
  double k, r0;
};

// overload serialization operator
mui::ostream& operator <<( mui::ostream& ost, const bond &v ) {
  // enumerate over primitive members
  return ost << v.k << v.r0;
}

// overload deserialization operator
mui::istream& operator >>( mui::istream& ist, bond &v ) {
  return ist >> v.k >> v.r0;
}
```

3.5. *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 situations, 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's dynamically typed data container. A frame is a collection of mappings from quantity names to actual MUI data container objects, while the frames themselves are again organized in a mapping where time stamps are used as indexing keys. This sparse storage structure, as illustrated in Fig. 4, 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.

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 Fig. 5 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.

The memory allocation for frames is managed transparently by a buffering scheme. The deallocation, however, must be set up by user because it is impossible to predict whether a frame will be reused in the future. Utility methods are provided for users 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 move semantic to avoid duplicate memory allocation or copying.

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. The multiple-program–multiple-data (MPMD) mode of MPI is a natural fit for the purpose of concurrent

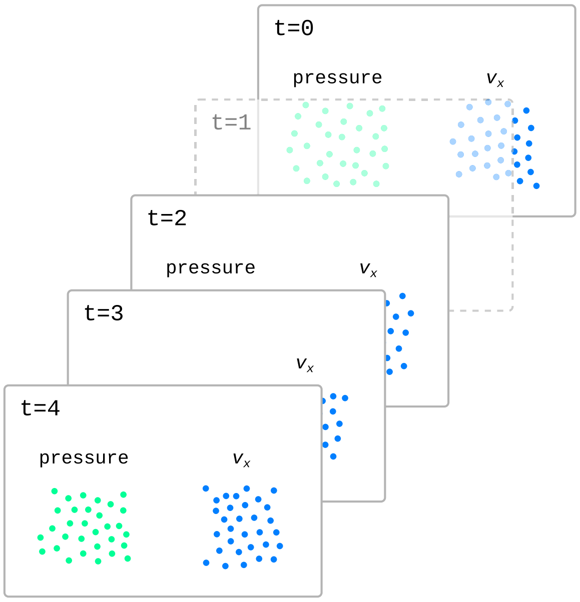

Fig. 4. Data points committed from different time steps are organized in time frames. Time frames can be non-uniformly distributed. Individual quantities can appear in a select subset, instead of all, of the time frames.

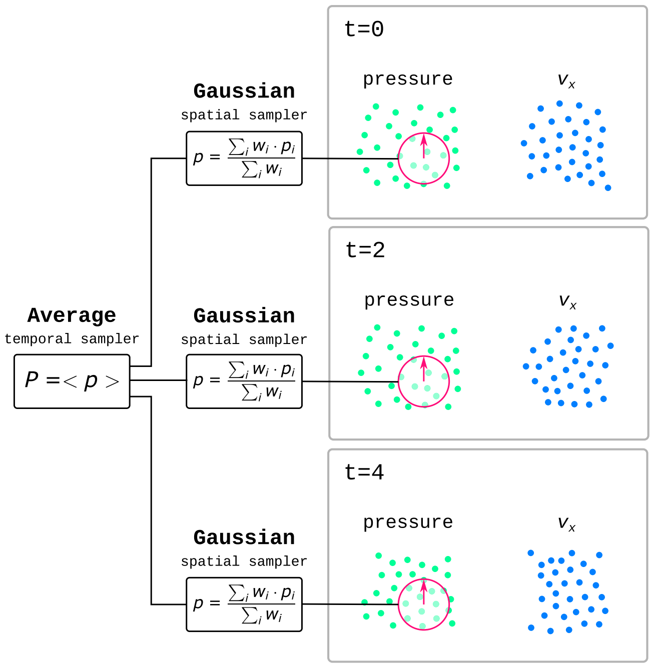

Fig. 5. Time samplers can interpolate or extrapolate the output of a spatial sampler over a range of time frames.

coupling. In this mode, each solver is compiled and linked separately as independent executables but invoked simultaneously as a single MPI job using the MPMD syntax. An example of the MPMD launch syntax is given in Listing 3. Theoretically, users can launch an arbitrary amount of ranks for each of the solvers irrespective of the number of ranks spawned for its peer solvers.

Listing 3 MPI MPMD launch syntax.

```
mpirun -np n1 solver1 solver1_arguments : -np n2 solver2 solver2_arguments
```

Algorithm 1 MPI MPMD setup.

```
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
```

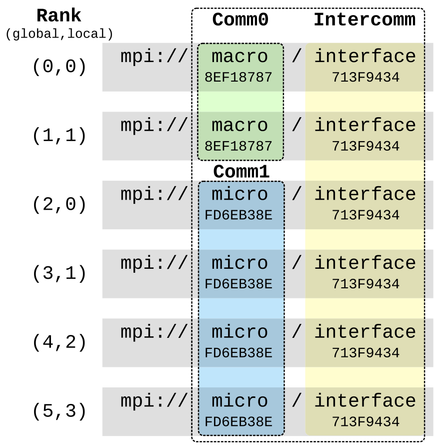
Rank
(global,local)
Comm0
Intercomm
(0,0) mpi:// macro 8EF18787 / interface 713F9434
(1,1) mpi:// macro 8EF18787 / interface 713F9434
Comm1
(2,0) mpi:// micro FD6EB38E / interface 713F9434
(3,1) mpi:// micro FD6EB38E / interface 713F9434
(4,2) mpi:// micro FD6EB38E / interface 713F9434
(5,3) mpi:// micro FD6EB38E / interface 713F9434

Fig. 6. In this example we demonstrate how an MPI job consisting of two macroscopic solver ranks and four microscopic solver ranks is partitioned according to the URI domain descriptor. The C++ **`std::hash`** functor can generate unique integer-valued hashes from the domain sub-string. MUI uses the hashes as the colors for splitting the MPI communicator. The hash value of the interface sub-string is then used to establish the inter-communicators that encompass both intra-communicators.

The MUI inter-solver communication topology is established dynamically from the MPI runtime job configuration. To ensure that this MPMD topology is only visible to MUI itself and hidden from the solver code, it is mandated by MUI that solvers should make no direct reference to the MPI predefined world communicator **`MPI_COMM_WORLD`** for any of its own communications. Instead, a globally accessible MPI communicator, *i.e.* a variable of type **`MPI_Comm`**, should be defined to hold a solver-specific global communicator, which can be obtained from a MUI helper function call that effectively splits the MPI predefined world communicator into subdomains using MPI application numbers. This is in fact one of the few modifications to the solvers that is ever dictated by MUI.

In order to identify and connect MUI instances belonging to different domains, each solver instance needs to initialize MUI using a uniform resource identifier (URI) domain descriptor with the format **`protocol://domain/interface`**. The **`protocol`** field must always be **`mpi`** in the current MUI implementation. It indicates that the inter-solver communication manager sends and receives messages through MPI. Internally, the communication managers are C++ objects allocated through an object factory mechanism, which would allow straightforward addition of new communication managers using alternative protocols such as TCP/IP or UNIX pipe. In our current communication scheme, the hash value of the **`domain`** sub-string is used as the *color* for splitting the MPI built-in world communicators into smaller ones each containing a single physical domain. The **`interface`** sub-string is also hashed to generate a unique integer value for identifying different interface regions in the case of simulating multi-interface systems. The actual algorithm, as given in Algorithm 1, makes use of the MPI **`MPI_COMM_SPLIT`** and **`MPI_INTERCOMM_CREATE`** method. Fig. 6 demonstrates the setup process in a system consisting of two subdomains and an interface between the subdomains.

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

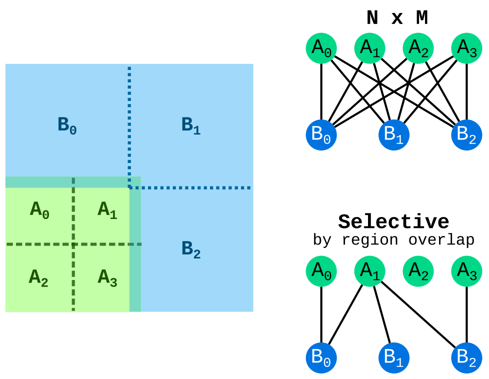

Fig. 7. An example illustrating the effect of smart sending. The green domain is handled by solver A and spatially decomposed between 4 MPI ranks, while the blue domain is handled by solver B using 3 MPI ranks. By checking for the overlap between the interface regions owned by the ranks, MUI can eliminate unnecessary data transfer between ranks such as $A_0 - B_1$, $A_2 - B_2$ and $A_3 - B_0$ etc. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Listing 4 The default configuration class for MUI.

```
struct crunch {
  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;
};
```

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. 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 Fig. 7, the regions are broadcasted among all the processes so that the communication between a sender and a receiver whose regions of push/fetch have 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 truly 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 the entire $\mathbb{R}^d$ with a validity period of infinity as a safety fall-back.

5. Customizability

MUI provides a vast customization space regarding communication content, spatial and temporal interpolation algorithm and communication pattern. In addition, MUI allows a number of its low-level traits to be parameterized at compile-time using a configuration class. A default configuration class **`crunch`** is shown in Listing 4. The class serves as the last template argument of all MUI component classes and samplers, and 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:

- dimensionality of the physical space;
- precision of floating point numbers;
- integer width;
- time stamp type;
- type list (as mentioned in Section 3.4);

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.

```
for $t = 0 : \delta t : T_{total}$ do
    ▷ Push
    for each particle $i$ do
        if WithinSendRegion($i$) then
            MUI::Push("$v_x$",coord[$i$],vel$_x$[$i$])
        end if
    end for
    MUI::Commit($t$)
    ▷ Fetch
    for each particle $i$ do
        if WithinReceiveRegion($i$) then
            $S_{spatial}$ ← Quintic($r$,$h$)
            $S_{temporal}$ ← ExactTime($\varepsilon$)
            vel$_x$[$i$] ← MUI::Fetch("$v_x$",coord[$i$],$t$,$S_{spatial}$,$S_{temporal}$)
        end if
    end for
    MUI::Forget($t$)
end for
```

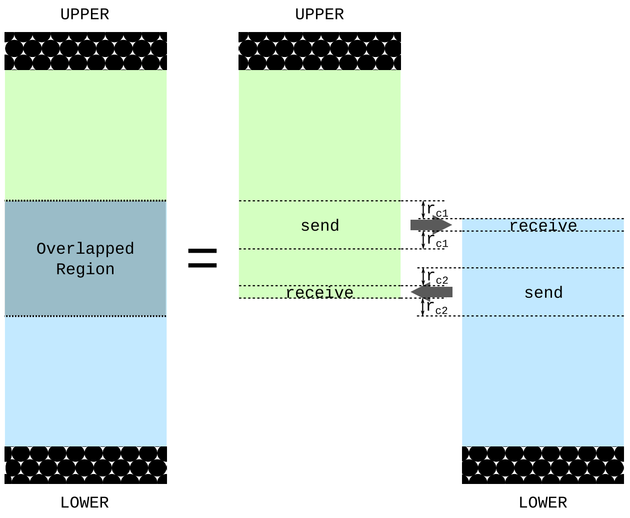

Fig. 8. The flow was simulated by two overlapping SPH domains. Each domain contains a send region and a receive region that are not overlapping with each other.

- debugging switch;
- exception handling behavior.

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

6. Demonstration examples

6.1. Couette flow: SPH–SPH coupling

To give a concrete example of the usage 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 Fig. 8, the system was simulated using two overlapping SPH domains, *i.e.* a lower one 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 [18] 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).

We then used the MUI-equipped 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

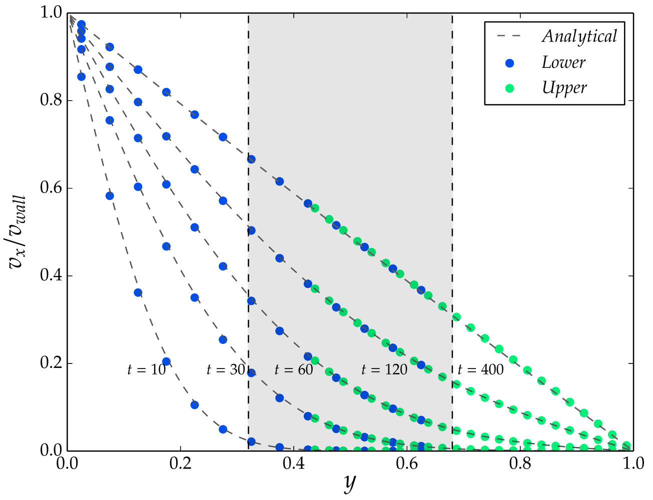

Fig. 9. Transient velocity profile of a Couette flow. The system was modeled by two SPH domains of different resolutions. Numerical solutions obtained from the lower and upper domains are plotted in blue and green, respectively. The analytic solutions are shown by dashed lines, while the interface region is indicated by the shaded box. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

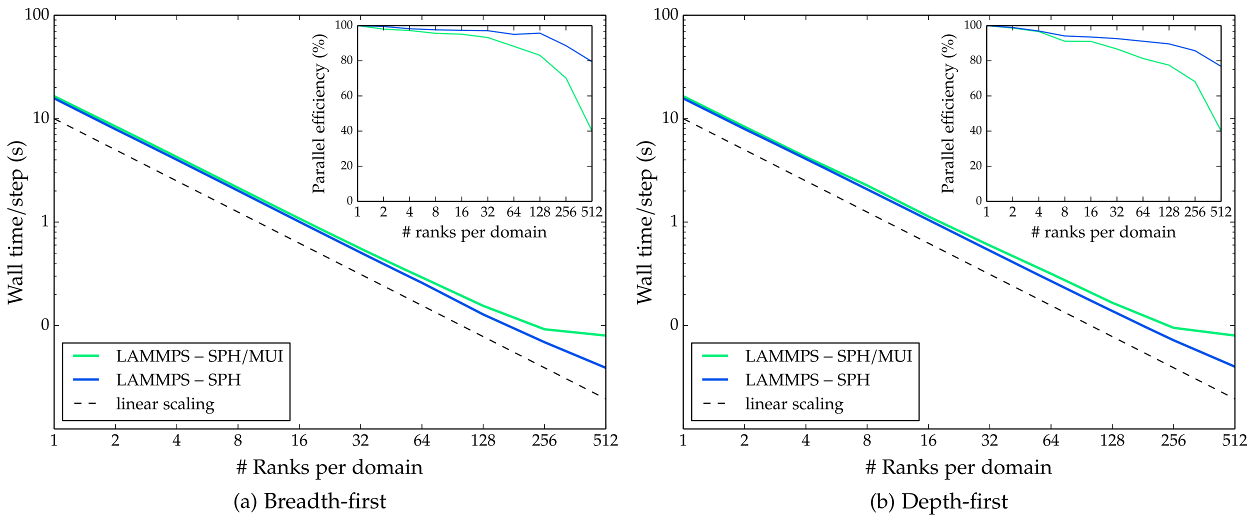

Fig. 10. Strong scaling performance comparison between the MUI-equipped and the original LAMMPS code. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

domains, respectively. As shown in Fig. 9 the velocity profile obtained from the coupled simulation is consistent with the analytic solution.

The performance and strong scalability of the MUI-equipped SPH solver were further characterized using a similar simulation setup. A system of size $6 \times 1 \times 6$ was simulated as a lower domain spanning from 0 to 0.6125 and an upper domain spanning from 0.3875 to 1. A number density of 40^3 was used for both the lower and upper subdomains for the ease of performance evaluation. Each domain thus contained 1 382 400 fluid particles and 172 800 wall particles. The computation was performed on the Eos supercomputer using up to 512 ranks for each subdomain on a total of 128 nodes each containing 2 Intel Xeon E5-2670 CPUs at 2.6 GHz. The asynchronous progress engine feature of Cray MPI was enabled to better accommodate the communication pattern of MUI. Two different rank placement strategies, *i.e.* breadth-first and depth-first, were used when increasing the amount of MPI ranks. With the breadth-first strategy, one MPI rank was spawned on each node until a maximum of 128 nodes were used. After that, the number of ranks per node was increased to further fill up the nodes until a total 1024 ranks were spawned. With the depth-first strategy we first increased the number of ranks per node up to 8 before adding in more nodes. Baseline performance metrics were obtained by simulating the lower and upper domains independently using the original LAMMPS solver.

Fig. 10 visualizes the measured scalability and parallel efficiency of the MUI-equipped LAMMPS versus the baseline solver as obtained from the benchmark. Fig. 11 presents a percentage breakdown of the CPU time spent in various parts of the solver at different levels of parallelism. In all cases, MUI consumes no more than 7% of the total CPU time. Note that this includes both the sampling time which is actually part of the useful work and hence should not be counted as overhead.

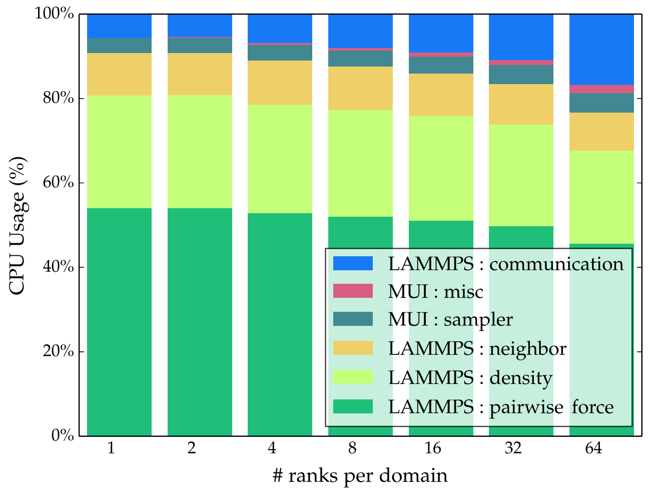

Fig. 11. Couette flow: a breakdown of the CPU time usage. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

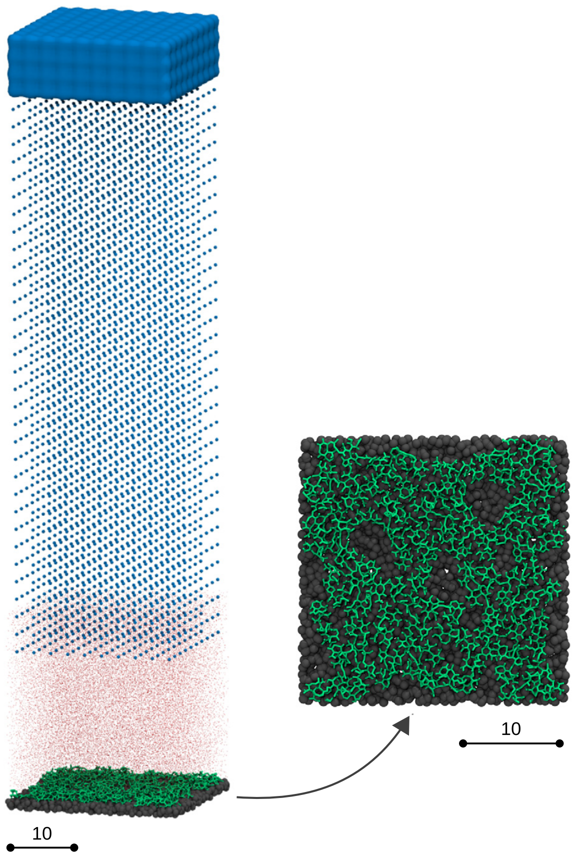

Fig. 12. A hybrid system consisting of an SPH upper domain and a DPD lower domain was modeled to study the hydrodynamical properties of dendrimer grafted surface. DPD wall particles, dendrimers, DPD solvent particles and SPH fluid particles are rendered in black, green, red and blue, respectively. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

6.2. Soft matter: SPH–DPD coupling

Next we demonstrate a concurrently coupled deterministic/stochastic simulation using a similar coupling scheme. As illustrated in Fig. 12, 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

Table 1
Soft matter: parameters for the SPH–DPD simulation.

DPD			SPH		
Arg	Val	Description	Arg	Val	Description
N_p	84 375	number of particles	N_p	4000	number of particles
ρ	5	particle number density	ρ_N	0.064	particle number density
γ	4.5	noise level	ρ_m	5	mass density
σ	3.0	dissipation	η	3.72	viscosity
δt	0.01	time step	δt	0.5	time step
r_c	1.0	cutoff distance	r_c	10	cutoff distance
$k_B T$	1.0	temperature level	c_s	1.5	speed of sound
g	0.001	body force	g	0.001	body force
$w_D = w_C^{0.5}$		weight function			

Table 2
Repulsive force constants a_{ij} for DPD.

	Wall	Dendrimer	Solvent
Wall	15	15	15
Dendrimer	15	15	75
Solvent	15	75	15

binary dendrimers. We used a 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 $40 \times 110 \times 40$ was constructed using an upper domain and a lower domain. The parameters for setting up the simulation are listed in Table 1. A gravity of 0.001 in the x direction was imposed for both domains.

The flow field in the upper domain with a simple no-slip boundary condition was simulated using the Smoothed Particle Hydrodynamics (SPH) method solving the Navier–Stokes equation. The SPH domain spanned 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.

The flow field in the lower domain was simulated using Dissipative Particle Dynamics (DPD) solving Newton's equation of motion in stochastic form. The DPD domain spanned from $y = -1$ to $y = 28$, and included 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 conservative force strength between different types of DPD particles are given in Table 2. The surface converage of the dendrimers was 70%. The viscosity of the DPD solvent was measured as 3.72.

The MUI-based coupling scheme is similar to that use 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 10 000 DPD units. As shown in Fig. 13 the effective channel width is reduced by the hydrophobic coating by about 0.5 DPD unit.

The simulation was done on a workstation with two quad-core Intel Xeon E5-2643 CPUs running at 3.30 GHz as well as four nVidia GeForce GTX TITAN GPUs each with 2688 cores. The DPD simulation was done on the four GPUs using the *USER*MESO package [19], while the SPH code ran on a single CPU core using the same LAMMPS SPH solver as mentioned in the previous example. This processor resource allocation was based on the observation that simulating the DPD domain is orders of magnitude more expensive than simulating the SPH domain.

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 [20] and the finite element method (FEM). The eDPD model is an extension to the classical DPD model [21,22] 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 α is the thermal diffusivity and $f = f_0 + f_{\text{interface}}$ 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. The eDPD solver then pushes the flux data back into the FEM solver. 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 boundaries were fixed at 270 K and 360 K as shown in Fig. 14. The thermal diffusivities of water and quartz were assumed to be constant at 0.143×10^6 m^2/s [23]

Algorithm 3 SPH–DPD coupling scheme. The C++ code for the **SumOver** sampler is given in Listing 4 in SI.

```
▷ SPH domain

for $t = 0 : 50\delta t : T_{total}$ do
    ▷ Push
    for each particle $i$ do
        if WITHINSENDREGION($i$) then
            MUI::PUSH("$v_x$",coord[$i$],vel$_x$[$i$])
        end if
    end for
    MUI::COMMIT($t$)
    ▷ Fetch
    for each particle $i$ do
        if WITHINRECEIVEREGION($i$) then
            $S_{spatial}$ ← QUINTIC($r_{DPD}, h_{DPD}$)
            $S_{temporal}$ ← SUMOVER($50\delta t$)
            vel$_x$[$i$] ← MUI::FETCH("$v_x$",coord[$i$],$t$,$S_{spatial}$,$S_{temporal}$)
        end if
    end for
    MUI::FORGET($t$)
end for

▷ DPD domain
for $t = 0 : \delta t : T_{total}$ do
    ▷ Push
    for each particle $i$ do
        if WITHINSENDREGION($i$) then
            MUI::PUSH("$v_x$",coord[$i$],vel$_x$[$i$])
        end if
    end for
    MUI::COMMIT($t$)
    ▷ Fetch
    $t_{SPH}$ ← FLOOR($t, 50\delta t$)
    for each particle $i$ do
        if WITHINRECEIVEREGION($i$) then
            $S_{spatial}$ ← QUINTIC($r_{SPH}, h_{SPH}$)
            $S_{temporal}$ ← EXACTTIME($\varepsilon$)
            vel$_x$[$i$] ← MUI::FETCH("$v_x$",coord[$i$],$t_{SPH}$,$S_{spatial}$,$S_{temporal}$)
        end if
    end for
    if MOD($t$,$50\delta t$) = 0 then
        MUI::FORGET($t - 50\delta t$)
    end if
end for
```

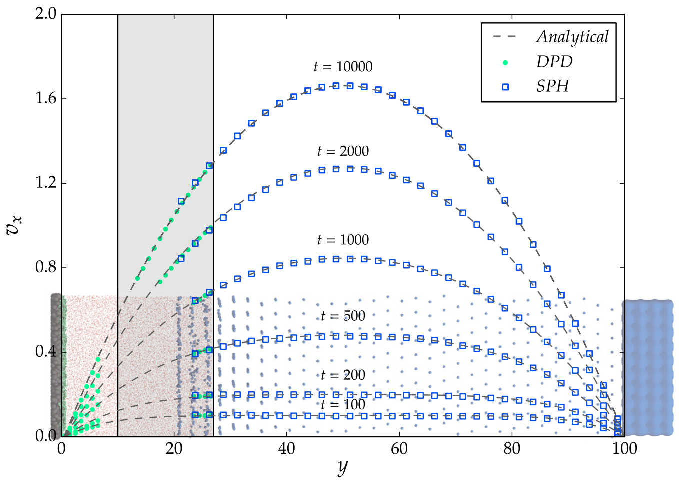

Fig. 13. The velocity profile derived from the coupled SPH/DPD simulation converges to that of a Poiseuille flow. SPH results, DPD results and the analytic solutions are plotted in blue dots, green dots and dashed line, respectively. The background picture indicates the system composition at the corresponding y position. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

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.

```
▷ eDPD domain
for $t = 0 : \delta t : T_{total}$ do
    ▷ Push
    $t_{FEM}$ ← Floor($t$, $10\delta t$)
    for each particle $i$ do
        if WithinCutoffOfCylinder($i$) then
            $S_{spatial}$ ← Linear($h_{max}$)
            $S_{temporal}$ ← ExactTime($\varepsilon$)
            $T_{wall}$ ← MUI::Fetch("$T$", coord[$i$], $t_{FEM}$, $S_{spatial}$, $S_{temporal}$)
            $q$ ← PerParticleHeatFlux(T[$i$], $T_{wall}$)
            MUI::Push("$q$", coord[$i$], $-q/C_v$)
        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 $t = 0 : 10\delta t : T_{total}$ 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
        $S_{spatial}$ ← VoronoiMean(Vertices)
        $S_{temporal}$ ← MeanOver($10\delta t$)
        $f_{interface}$[$i$] ← MUI::Fetch("$q$", coord[$i$], $t$, $S_{spatial}$, $S_{temporal}$)
    end for
    MUI::Forget($t$)
    SolveForNextStep
end for
```

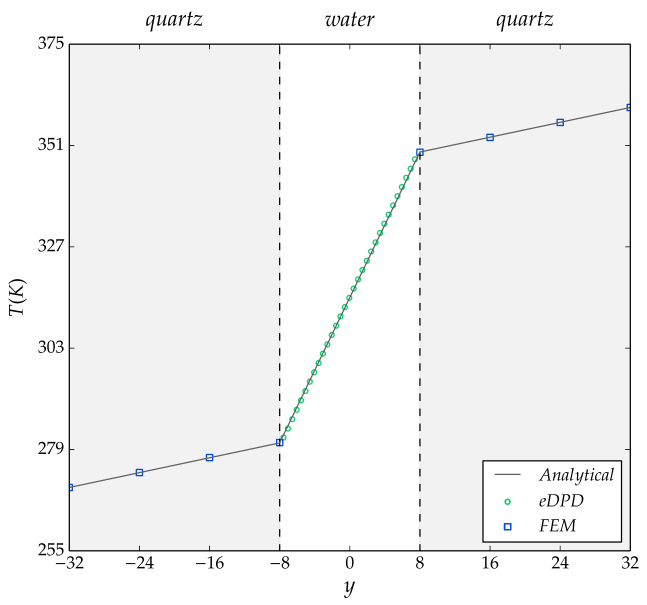

Fig. 14. Heat conduction: temperature profile obtain for a quartz–water–quartz tri-layer system with the FEM solver handling the solid domain and the eDPD solver handling the fluid domain.

and 1.4×10^6 m^2/s [24] over the temperate range, respectively, while the interfacial thermal diffusivity was chosen as the arithmetic mean between the two values.

Table 3
Conjugate heat transfer: parameters for the eDPD–FEM simulation of immersed heating cylinder are taken from Ref. [20].

Length scale		$L_0 = 11$ nm	
Time scale		$\tau = 0.935$ ns	
Temperature scale		$T_0 = 300$ K	
Mass scale		$m_0 = 3.32 \times 10^{-22}$ kg	
eDPD		FEM	
Arg	Val	Arg	Val
α	1.43×10^{-7} m^2/s	α	1.43×10^{-7} m^2/s
δt	0.0125τ	δt	0.125τ
ν	8.57×10^{-7} m^2/s	f_0	$0.004T_0/\tau$
		space	P_1

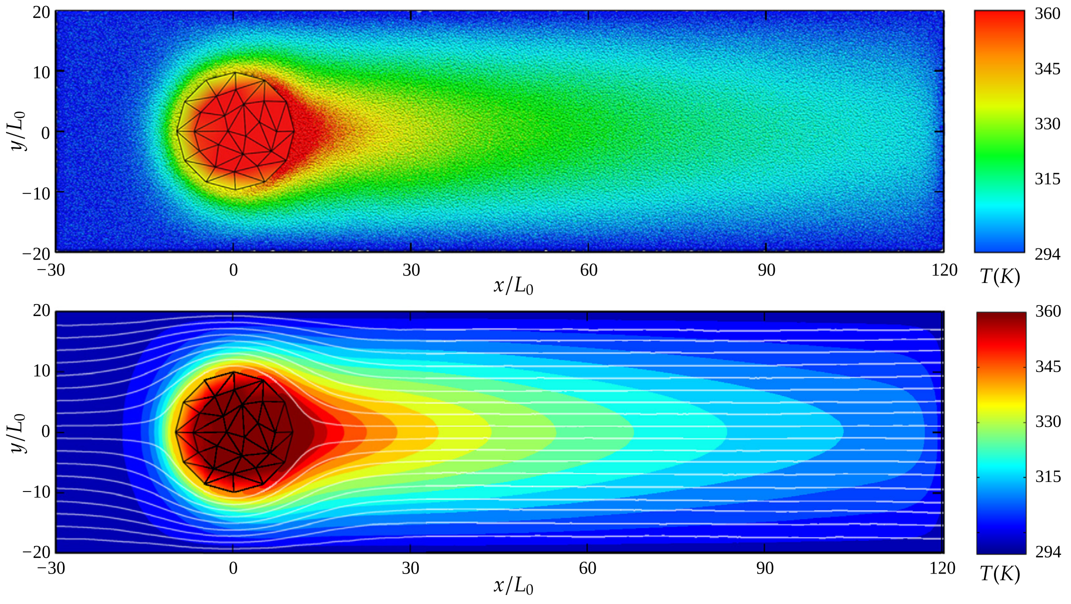

Fig. 15. Conjugate heat transfer: a snapshot of the hybrid particle/mesh structure of the domain at steady state is shown in the upper plot; streamlines and the temperature field in the system at steady state are visualized in the lower plot by white lines and color-mapped contours, respectively. (For interpretation of the colors in this figure, the reader is referred to the web version of this article.)

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 3. The simulation result is shown in Fig. 15. The Reynolds number is defined by $Re = (v_{max}D)/\nu = 1.97$ where $v_{max} = 0.65L_0/\tau$ is the maximum inlet velocity, $\nu = 6.62$ the kinematic viscosity and $D = 20.0L_0$ 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.

The simulation was performed on a workstation with two hexa-core Intel Xeon E5-2630L CPUs running at 2.0 GHz. The eDPD simulation occupied 11 CPU cores using our customized LAMMPS code. The FEM solver ran on a single CPU core. This computational resource configuration was based the relative computational cost of the two codes.

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, universality, 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 [10–12] appears to be the most widely adopted standard for this types of coupling with conforming implementations such as Uintah [13–15], CCAT [25], SCIRun2 [26] and so on [27,28]. 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. Compared to MUI, the interfaces exposed by CCA-based frameworks typically require more engineering effort to cover lower level program activities such as inter-rank communication protocol handling, load balancing and memory management. There exist also coupling frameworks that require more extensive modification of existing solvers. Solving algorithms are encapsulated as classes that expose a given set of interfaces in the Kratos [29] framework. The DDEMA [30] framework focuses on PDE solvers using mesh-based methods and casts software packages into an *actor* design pattern. MUI, on the contrary, does not require code refactoring and only requires the insertion of a few lines of pushing/fetching code.

In addition, various specialized coupling frameworks have been proposed. For example, the PPM library [31] represents one of the major efforts in carrying out simulations with a unified particle and mesh representation. 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 [32]. It is essentially a mathematical framework that formulates the way in which information should be exchanged at subdomain boundaries. In fact, this algorithm can be implemented with MUI in a straightforward manner. The Macro–Micro-Coupling [33,34] framework can solve coupling problems between macroscopic models and microscopic models. The MCI [35] framework is able to couple massively parallel spectral-element simulations and particle-based simulations in a highly efficient and scalable manner. The MUPHY [36] framework couples Lattice-Boltzmann method with molecular dynamics simulation. The MUSE [37,38] framework is specialized in astrophysics simulation. In contrast, MUI is able to accommodate any method because it allows the encoding of solver-/method-specific information as data points of arbitrary custom value. It can facilitate 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 delivers excellent parallel efficiency and can be adopted easily for coupling heterogeneous simulations. It is intended to be a development tool that can be used for fast implementation of abstract mathematical frameworks and coupling methodologies.

Acknowledgements

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). Simulations were carried out at the Oak Ridge Leadership Computing Facility through the Director Discretion project BIP102 and the Innovative and Novel Computational Impact on Theory and Experiment program BIP118. Particle simulation results were visualized with VMD [39,40].

Appendix A

A.1. Language compatibility

The main body of MUI is written in C++11, the version of ISO standard C++ ratified in 2011. Due to the extensive usage of template programming techniques and C++11 newly-introduced features, the code has to be compiled with more recent versions of C++ compilers that have complete C++11 support. Table 4 summarizes our experimental result on the versions of common compilers which were able to build MUI-enabled solvers. The list is conservative in that older compilers may also work depending on the specific situation. To compile with C++11, a single compiler option, *e.g.* `-std=c++11` for GCC, has to be added into the compiler command line. A wrapper for MUI is needed for projects developed in other languages. A sample C wrapper and a sample Fortran wrapper are included in MUI, while adaptations to other languages can be made relatively easily.

Table 4
Compilers compatibility of MUI. For those not currentlyDepartment of Energy (DoE) Collaboratory on Mathematics for Mesoscopic Modeling of Materials (CM4)supporting MUI, full C++11 feature support was also announced and will be available within the near future.

Compiler	Maintainer	Version	Flag
GCC	GNU	4.8.3	`-std=c++11`
Clang	Apple	3.5.0	`-std=c++11`
Intel C++	Intel	15.0	`-std=c++11`, `/Qstd=c++11`
NVCC	nVIDIA	7.0	`-std=c++11`
Visual C++	Microsoft	–	
XL C++	IBM	–	
PGCC	PGI	–	

Appendix B. Supplementary material

Supplementary material related to this article can be found online at http://dx.doi.org/10.1016/j.jcp.2015.05.004.

References

[1] M. Praprotnik, L.D. Site, K. Kremer, Multiscale simulation of soft matter: from scale bridging to adaptive resolution, Annu. Rev. Phys. Chem. 59 (2008) 545–571.
[2] K. Mohamed, A. Mohamad, A review of the development of hybrid atomistic–continuum methods for dense fluids, Microfluid. Nanofluid. 8 (3) (2010) 283–302.
[3] E. Weinan, Principles of Multiscale Modeling, Cambridge University Press, 2011.
[4] P.-L. Lions, On the Schwarz alternating method. I, in: First International Symposium on Domain Decomposition Methods for Partial Differential Equations, Paris, France, 1988, pp. 1–42.
[5] J. Xu, Iterative methods by space decomposition and subspace correction, SIAM Rev. 34 (4) (1992) 581–613.
[6] J.H. Walther, M. Praprotnik, E.M. Kotsalis, P. Koumoutsakos, Multiscale simulation of water flow past a C540 fullerene, J. Comput. Phys. 231 (7) (2012) 2677–2681.
[7] Z. Chen, S. Jiang, Y. Gan, H. Liu, T.D. Sewell, A particle-based multiscale simulation procedure within the material point method framework, Comput. Part. Mech. 1 (2) (2014) 147–158.
[8] R. Delgado-Buscalioni, K. Kremer, M. Praprotnik, Concurrent triple-scale simulation of molecular liquids, J. Chem. Phys. 128 (11) (2008) 114110.
[9] A. Patronis, D.A. Lockerby, Multiscale simulation of non-isothermal microchannel gas flows, J. Comput. Phys. 270 (2014) 532–543.
[10] B.A. Allan, R.C. Armstrong, A.P. Wolfe, J. Ray, D.E. Bernholdt, J.A. Kohl, The CCA core specification in a distributed memory SPMD framework, Concurr. Comput.: Pract. Exp. 14 (5) (2002) 323–345.
[11] S. Lefantzi, J. Ray, H. Najm, Using the common component architecture to design high performance scientific simulation codes, in: Parallel and Distributed Processing Symposium, 2003. Proceedings. International, April 2003, pp. 10.
[12] L. McInnes, 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, S. Zhou, Parallel PDE-based simulations using the common component architecture, in: A. Bruaset, A. Tveito (Eds.), Numerical Solution of Partial Differential Equations on Parallel Computers, in: Lecture Notes in Computational Science and Engineering, vol. 51, Springer, Berlin, Heidelberg, 2006, pp. 327–381.
[13] J. Davison de St. Germain, J. McCorquodale, S. Parker, C. Johnson, Uintah: a massively parallel problem solving environment, in: High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on, 2000, pp. 33–41.
[14] S.G. Parker, A component-based architecture for parallel multi-physics PDE simulation, Future Gener. Comput. Syst. 22 (1–2) (2006) 204–216.
[15] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C.A. Wight, J.R. Peterson, Uintah: a scalable framework for hazard analysis, in: Proceedings of the 2010 TeraGrid Conference, TG '10, ACM, New York, NY, USA, 2010, pp. 3:1–3:8.
[16] B. Allan, S. Lefantzi, J. Ray, ODEPACK++: refactoring the LSODE 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, April 2004, pp. 109–119.
[17] J.F. Hughes, S.K. Feiner, J.D. Foley, K. Akeley, M. McGuire, A. van Dam, D.F. Sklar, Computer Graphics: Principles and Practice, 2013.
[18] S. Plimpton, P. Crozier, A. Thompson, LAMMPS-large-scale atomic/molecular massively parallel simulator, Sandia National Laboratories, 2007.
[19] Y.-H. Tang, G.E. Karniadakis, Accelerating dissipative particle dynamics simulations on GPUs: algorithms, numerics and applications, Comput. Phys. Commun. 185 (11) (2014) 2809–2822.
[20] Z. Li, Y.-H. Tang, H. Lei, B. Caswell, G.E. Karniadakis, Energy-conserving dissipative particle dynamics with temperature-dependent properties, J. Comput. Phys. 265 (2014) 113–127.
[21] P.J. Hoogerbrugge, J.M.V.A. Koelman, Simulating microscopic hydrodynamic phenomena with dissipative particle dynamics, Europhys. Lett. 19 (3) (1992) 155.
[22] P. Español, P. Warren, Statistical mechanics of dissipative particle dynamics, Europhys. Lett. 30 (4) (1995) 191.
[23] J. Blumm, A. Lindemann, Characterization of the thermophysical properties of molten polymers and liquids using the flash technique, High Temp., High Press. 35 (36) (2007) 6.
[24] B. Gibert, D. Mainprice, Effect of crystal preferred orientations on the thermal diffusivity of quartz polycrystalline aggregates at high temperature, Tectonophysics 465 (1) (2009) 150–163.
[25] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, M. Yechuri, A component based services architecture for building distributed applications, in: High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on, IEEE, 2000, pp. 51–59.
[26] K. Zhang, K. Damevski, V. Venkatachalapathy, S. Parker, SCIRun2: a CCA framework for high performance computing, in: High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on, April 2004, pp. 72–79.
[27] D. Gannon, S. Krishnan, L. Fang, G. Kandaswamy, Y. Simmhan, A. Slominski, On building parallel grid applications: component technology and distributed services, in: Challenges of Large Applications in Distributed Environments, 2004. CLADE 2004. Proceedings of the Second International Workshop on, June 2004, pp. 44–51.
[28] S. Li, P. Mulunga, Q. Yang, X. Sun, A common component architecture (CCA) based design and implementation for distributed parallel magnetotellurice forward model, in: Information Management, Innovation Management and Industrial Engineering (ICIII), 2013 6th International Conference on, vol. 3, Nov. 2013, pp. 433–436.

[29] P. Dadvand, R. Rossi, M. Gil, X. Martorell, J. Cotela, E. Juanpere, S. Idelsohn, E. Oñate, Migration of a generic multi-physics framework to HPC environments, in: Selected contributions of the 23rd International Conference on Parallel Fluid Dynamics, ParCFD2011, Comput. Fluids 80 (2013) 301–309.
[30] J. Michopoulos, P. Tsompanopoulou, E. Houstis, J. Rice, C. Farhat, M. Lesoinne, F. Lechenault, DDEMA: a data driven environment for multiphysics applications, in: P. Sloot, D. Abramson, A. Bogdanov, Y. Gorbachev, J. Dongarra, A. Zomaya (Eds.), Computational Science, ICCS 2003, in: Lecture Notes in Computer Science, vol. 2660, Springer, Berlin, Heidelberg, 2003, pp. 309–318.
[31] I.F. Sbalzarini, J.H. Walther, M. Bergdorf, S.E. Hieber, E.M. Kotsalis, P. Koumoutsakos, PPM – a highly efficient parallel particle–mesh library for the simulation of continuum systems, J. Comput. Phys. 215 (2) (2006) 566–588.
[32] D.A. Fedosov, G.E. Karniadakis, Triple-decker: interfacing atomistic–mesoscopic–continuum flow regimes, J. Comput. Phys. 228 (4) (2009) 1157–1171.
[33] P. Neumann, J. Harting, Massively parallel molecular–continuum simulations with the Macro–Micro-Coupling tool, in: Hybrid Particle-Continuum Methods in Computational Materials Physics, vol. 46, 2013, pp. 211.
[34] P. Neumann, W. Eckhardt, H.-J. Bungartz, Hybrid molecular–continuum methods: from prototypes to coupling software, in: Mesoscopic Methods for Engineering and Science, Proceedings of ICMMES-2012, Taipei, Taiwan, 23–27 July 2012, Comput. Math. Appl. 67 (2) (2014) 272–281.
[35] L. Grinberg, V. Morozov, D. Fedosov, J. Insley, M. Papka, K. Kumaran, G. Karniadakis, A new computational paradigm in multiscale simulations: application to brain blood flow, in: High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for, Nov. 2011, pp. 1–12.
[36] M. Bernaschi, S. Melchionna, S. Succi, M. Fyta, E. Kaxiras, J. Sircar, MUPHY: a parallel MUlti PHYsics/scale code for high performance bio-fluidic simulations, Comput. Phys. Commun. 180 (9) (2009) 1495–1502.
[37] S. Portegies Zwart, S. McMillan, S. Harfst, D. Groen, M. Fujii, B.Ó. Nualláin, E. Glebbeek, D. Heggie, J. Lombardi, P. Hut, et al., A multiphysics and multiscale software environment for modeling astrophysical systems, New Astron. 14 (4) (2009) 369–378.
[38] S.F.P. Zwart, S.L. McMillan, A. van Elteren, F.I. Pelupessy, N. de Vries, Multi-physics simulations using a hierarchical interchangeable software interface, Comput. Phys. Commun. 184 (3) (2013) 456–468.
[39] W. Humphrey, A. Dalke, K. Schulten, VMD – visual molecular dynamics, J. Mol. Graph. 14 (1996) 33–38.
[40] J. Stone, An efficient library for parallel ray tracing and animation, Master's thesis, Computer Science Department, University of Missouri-Rolla, April 1998.