Experiments with Memory-to-Memory Coupling for End-to-End Fusion Simulation Workflows

Ciprian Docan
Fan Zhang
Manish Parashar
Center for Autonomic Computing
Rutgers University, Piscataway NJ, USA
{docan,zhangfan,parashar}@cac.rutgers.edu

Julian Cummings
California Institute of Technology
1200 E. California Blvd.
Pasadena, CA 91125, USA
cummings@cacr.caltech.edu

Norbert Podhorszki
Scott Klasky
Oak Ridge National Laboratory
P.O. Box 2008, Oak Ridge, TN, 37831, USA
{pnorbert,klasky}@ornl.gov

Abstract—Scientific applications are striving to accurately simulate multiple interacting physical processes that comprise complex phenomena being modeled. Efficient and scalable parallel implementations of these coupled simulations present challenging interaction and coordination requirements, especially when the coupled physical processes are computationally heterogeneous and progress at different speeds.

In this paper, we present the design, implementation and evaluation of a memory-to-memory coupling framework for coupled scientific simulations on high-performance parallel computing platforms. The framework is driven by the coupling requirements of the Center for Plasma Edge Simulation, and it provides simple coupling abstractions as well as efficient asynchronous (RDMA-based) memory-to-memory data transport mechanisms that complement existing parallel programming systems and data sharing frameworks. The framework enables flexible coupling behaviors that are asynchronous in time and space, and it supports dynamic coupling between heterogeneous simulation processes without enforcing any synchronization constraints. We evaluate the performance and scalability of the coupling framework using a specific coupling scenario, on the Jaguar Cray XTS system at Oak Ridge National Laboratory.

Keywords—coupling; memory-to-memory; rdma; workflows

I. INTRODUCTION

Scientific simulations are constantly striving to achieve highly accurate numerical solutions for realistic models of complex physics phenomena using very large-scale, high-performance parallel machines. An increasing number of these applications simulate multiple interacting physical processes that comprise the complex phenomena being modeled. Efficient and scalable parallel implementations of these coupled simulations present challenging requirements in terms of the orchestration and coordination requested between independently running codes that have to exchange data (parameters or variables) at runtime, especially when the coupled physical processes are computationally heterogeneous and progress at different speeds. For example, computationally more intensive models may run on a larger number of processors, while validating models may run on a smaller number of processors. Furthermore, coupling may also require data from a first simulation to be processed before it is fed to a second simulation. A key requirement for achieving performance, scalability and system efficiency in these cases is that the coordinations and interactions are asynchronous and decoupled, both in space and time.

Many existing solutions for supporting parallel coupled simulations are based on the underlying parallel programming frameworks used by the simulations, such as MPI or SHMEM. These frameworks are typically optimized for the hosting platforms and can potentially deliver high performance. However, these solutions only work well when the coupling behaviors can be pre-defined, and the coupled processes are computationally homogeneous, well balanced and can be synchronized (e.g., exchange data at the same time with all processors involved in the exchange) without significant penalties. Furthermore, implementing general couplings that require data manipulation can be non-trivial. Workflow frameworks provide an alternate coupling approach using files to exchange data between the coupled processes. While this method supports loose couplings, it is limited in the type and flexibility of the coordination behaviors, interactions and data exchanges that are supported (e.g., files). Furthermore, it can introduce overheads due to the file system, and is not optimized to exploit the high-bandwidth/low-latency data channels available in high-performance computing platforms, such as, fast direct memory access between processors.

This paper presents the design, implementation and evaluation of a coupling framework that provides the abstractions and mechanisms to efficiently and scalably support asynchronous memory-to-memory coupling on high-performance parallel computing platforms. The coupling framework builds on the DART [1] asynchronous data transport substrate, which uses RDMA (Remote Direct Memory Access) support provided by advanced communication technologies and is optimized for fast and asynchronous data transfers, with low latency and small overhead on applications. Furthermore, it enables direct memory-to-memory communication between processing nodes of distinct simulations using RDMA and the overlap of computations and communications, allowing for a better utilization of the computing resources.

The coupling framework is composed of two key components. A client side component is integrated with the coupled simulations and provides them with an abstraction of a virtual shared space as well as a simple put/get API for accessing
this space. The second component runs on a small number of separate nodes providing a transient in-memory temporary staging area for the exchanged data. Data put by one process is indexed, processed and stored in the staging area, and it can be asynchronously read by other processes using the get operator. The framework ensures the integrity and coherence of the shared data and manages garbage collection. Internally, the framework implements data placement and load balancing policies, and it provides functions for data query and retrieval. Note that the coupling framework can co-exist with other parallel programming systems (e.g., MPI or SHMEM). Using the framework, coupled processes can progress independently at different rates, and exchange data at runtime, without making any assumptions about the frequencies of interactions or the relative execution speeds of the codes or forcing any synchronizations. The data exchange is transparent and independent of the parallelization or partitioning of the coupled processes. The coupling framework has been implemented and deployed on the Jaguar Cray XT5 system at Oak Ridge National Laboratory and is being used to support coupled fusion simulation workflows as part of the Center for Plasma Edge Simulation, a US Department of Energy prototype Fusion Simulation Project (FSP) center. The experimental evaluation presented in this paper demonstrates its performance as well as its scalability.

The rest of the paper is structured as follows. Section II describes the driving fusion simulation application as well as its coupling requirements. Section III describes the design and implementation of the memory-to-memory coupling framework, and Section IV describes its use to support the coupled fusion simulation workflow. Section V presents the experimental evaluation. Section VI presents related work and Section VII concludes the paper.

II. DRIVING APPLICATIONS: COUPLED FUSION SIMULATIONS

The coupled fusion simulations, which are the driving applications for the coupling framework described in this paper, are developed as part of the Center for Plasma Edge Simulation (CPES). The overall goal of CPES is to develop a new integrated predictive plasma edge simulation package applicable to existing magnetic fusion facilities and next-generation burning plasma experiments such as ITER (International Thermonuclear Experimental Reactor).

A. Problem Description

A tokamak fusion reactor typically has very hot, dense plasma in the core region that is confined by nested magnetic surfaces and inside which the fusion reactions take place. The core plasma is heated initially by external means and then later by the fusion reactions themselves (i.e., a burning plasma), and this energy leaks out slowly to the edge plasma region mostly via turbulent transport processes. A critical goal of fusion experiments is to reduce this radial cross-field transport and improve energy confinement to the point where a burning plasma can become self-sustaining for prolonged periods of time. Previous experiments have shown that strong core plasma heating can produce a thin layer of almost turbulence-free plasma in the edge region known as the H-mode layer. The reduction in turbulent transport, along with kinetic effects in the unique magnetic geometry of the tokamak edge region, leads to a steep local plasma pressure gradient and the development of a high pedestal in the edge plasma density and temperature profiles. This pedestal is a beneficial feature that increases fusion reaction rates in the core plasma by raising the central density and temperature.

The edge pedestal height and width appear to be limited by the development of magnetohydrodynamic (MHD) instabilities known as edge localized modes (ELMs) that are triggered by steep pressure gradients or strong edge plasma currents. Not only do ELMs thus constrain the production of fusion power in the core plasma, they also produce a rapid and sometimes violent ejection of heat and plasma particles onto the material walls of the fusion reactor. The success of burning plasma experiments such as ITER depends heavily upon achieving H-mode operation with an edge pedestal of sufficient height without triggering large-scale ELMs. Hence, the development of a predictive capability for the edge plasma pedestal and the accompanying ELMs is a high priority in fusion plasma research.

A complete understanding of the edge pedestal structure requires a full distribution function kinetic simulation due to the presence of a steep pressure gradient, low plasma collisionality in the edge, particle orbit squeezing effects, and a non-Maxwellian ion distribution. The kinetic code must include neoclassical effects (i.e., transport due to collisions in an inhomogeneous magnetic field) on a dynamical system of ions, electrons and neutrals, as well as self-consistent electromagnetic perturbations and turbulence. On the other hand, the onset and nonlinear evolution of ELM instabilities is best understood in the context of a reduced or two-fluid MHD model. One can use an MHD equilibrium solver to compute kinetic modifications to the initial fluid equilibrium. An ideal MHD linear analysis code can be applied to compute eigenmodes and linear growth rates and check for any strongly unstable ELMs. Finally, a nonlinear MHD simulation of unstable ELMs can be performed to track the effects of the ELMs on the plasma profile until a new MHD equilibrium emerges. A clear picture of the interplay between edge pedestal buildup and ELMs can thus be most efficiently obtained through a coupled simulation system that links a kinetic edge code with a proper set of MHD analysis codes.

B. Code Coupling Requirements in Fusion Simulations

Modern experiments in magnetic confinement fusion such as the DIII-D tokamak reactor at General Atomics combine a very hot, dense and turbulent ionized gas or plasma, applied and self-generated magnetic fields in a specific configuration for plasma confinement, and neutral particle beams and radio frequency (RF) waves for heating the plasma. The result is an extremely rich physical system that encompasses atomic physics, nuclear reactions, wave and charged particle dynam-
ics, electromagnetics and turbulence, with important physical effects occurring on a wide variety of spatial and temporal scales. It is well known that there can be strong interactions and coupling effects between the various physical phenomena taking place on different scales or at different physical locations with the reactor device. For example, RF heating can strongly modify the plasma particle distribution in velocity space, thus affecting plasma collisionality and the self-generated current profile, which in turn will modify plasma confinement. Nevertheless, the complexity of the physics along with available computing capacity have until very recently limited most fusion simulation models to addressing one specific physical aspect of a tokamak experiment operating within a very restricted spatial and temporal range. Thus, we have computational models for RF heating, neutral beam heating, MHD instabilities, microturbulence, plasma-surface interactions, particle and heat radial transport, and so on.

From an implementation point of view, these coupling requirements present multiple challenges, in terms of designing and managing appropriate and physically meaningful data exchanges between simulation models and finding ways to effectively bridge large gaps in the valid spatial and temporal scales for these models.

III. A MEMORY-TO-MEMORY COUPLING FRAMEWORK

A. System Architecture

The memory-to-memory coupling framework enables dynamic coupling and data sharing at runtime between heterogeneous simulations. Its architecture consists of two main components as shown in Fig. 1, a Client Component and a Server Component. Conceptually, the framework provides simulations with an abstraction of a virtual shared space, hosted by the server component, that enables them to share data by asynchronously inserting and retrieving data objects using the client component. The two components are described below.

The Client and Server components build on a common data transport layer, i.e., the DART RPC Layer. The DART RPC layer is based on the DART [1] asynchronous data transport substrate and provides the data communication primitives used by the upper layers. The current implementation of DART is based on RDMA technology and specifically, the Portals [2] RDMA library. However, the transport layer can be efficiently ported to other RDMA enabled systems such as InfiniBand, DCMF, iWarp, making the DART framework portable to other systems. The asynchronous RPC services provided by this layer are very flexible and can easily be extended, i.e., the upper layers can add new services by registering a callback function pointer and a service identifier. Services can be dynamically added, removed and accessed at runtime.

**Client Component**: The client component is integrated with the coupled simulation codes and provides the abstractions and APIs for coupling. Simulations share in-memory data objects that behave like files, and the API provided is similar to file operations, e.g., open(), read(), write() and close(), but operates on these in-memory data objects. Note that a data object is an abstraction for the unit of data sharing and may be composed of heterogeneous data types.

The **DART Client Layer** provides services to register and deregister a simulation instance with the server component, and to discover other peer clients in the system. The **DART Client Object Layer** provides services to operate on data objects, e.g., insert to write an object to, or retrieve to request an object from the server component. It also provides services to create and access the data objects in memory.

This layer is responsible for load balancing the shared data objects across the staging nodes that host the DART server component. It addresses two aspects of load balancing, balancing the data storage and the metadata accesses. Initial balancing of data storage is performed during the registration process where the client instances are distributed equally among the server instances. At runtime, each client component stores its data objects at its associated server instance, and as a result, application data is evenly distributed across the server instances. Metadata accesses are balanced by dynamically choosing a server component using a hashing mechanism based on the data object identifier. A corresponding hashing function is used to locate the data object in the server component during a retrieve operation.

**Server Component**: The server component runs independently on a dynamic set of staging nodes, and it provides transient in-memory storage, implemented as a distributed hash table (DHT), for the exchanged data objects. It provides its own mechanism for server peers discovery and registration in the system. The **DART Server Object Layer** provides services for sharing data objects, i.e., retrieve and insert, which complement the corresponding abstractions provided by the client component. Note that the server component is an internal component in that it does not provide any direct API to applications. The **Storage Layer** implements the transient in-memory storage for inserted data objects until they are retrieved. Internally this layer provides a garbage collection mechanism, which releases resources associated with data objects that are no longer referenced. It also implements a coherency protocol that ensures data integrity and consistency.
B. Framework Operation

The operation of the memory-to-memory coupling framework requires that the server component is started and initialized first. When the simulations start, they have to explicitly initialize their client components, which know how to register the simulation instances and discover peer nodes in the server component. The registration phase is part of a discovery mechanism, which enables the client component to subsequently share data objects directly in a peer-to-peer manner.

A simulation can use the API provided by the client component to create locally and share a data object. To create a data object, it opens the object, creates its content by writing to it and finally closes the object. When the data object is closed, the client component transparently inserts it into the server component. The simulation can also specify the number of copies/instances for a data object that are inserted. Similarly, a simulation can retrieve a data object. The client component transparently retrieves a copy of the data object, which the simulation can then access using the API provided, i.e., it opens the data object, reads its content, and closes it. When a data object is closed, the client component destroys it and releases the associated resources.

A server component stores shared data objects in its storage layer, and the simulation nodes can asynchronously retrieve them as needed. Because the framework is distributed, a coherency protocol is implemented to ensure consistency. This protocol manages consistent access to the shared objects and determines when a data object can be inserted, destroyed, released or replaced. The first copy of a new data object is simply added to the storage; subsequent copies of the same data object can either be added to the storage, or can replace existing objects that have not been used or consumed by any simulation node. A new object can replace an old object if the old object has zero remaining copies available or if it has not been retrieved at all. If, upon a request, a simulation node retrieves one instance of an object, then the coherency protocol guarantees that all the other simulation nodes will receive the same instance of the object.

When a simulation requests a data object, the server component serves one copy of the data object if it already exists in this storage layer, or else it blocks the simulation until it receives a copy for the requested object. A blocked retrieve operation is transparent to a simulation, and the client component automatically resumes execution when the data object becomes available at the server component. This provides a simple synchronization mechanism for simulations that may progress at different speeds and have to exchange data objects at runtime.

The server component runs on a smaller number of nodes as compared to the simulations, and one server node typically serves multiple simulation nodes. The number of communication buffers for direct RDMA messages at each server node is limited, and because the requests from the client component are driven by the simulation, they can easily overflow the number of buffers available. The DART Server Layer and the DART Client Layer implement an automatic flow control mechanism, which is based on flow updates that are piggybacked on exchanged messages.

The overall framework implementation is single-threaded, and it is request-driven on the server component side and simulation-driven on the client component side. However, a blocked retrieve request does not block the simulation and allows for other requests to proceed because the RPC implementation uses callback pointers and is re-entrant. This is very important, especially since the server component has to service multiple concurrent requests from client nodes.

IV. Case Study: CPES Full-ELM Coupling

A. CPES Full-ELM Coupling Workflow

The CPES Full-ELM coupling application workflow [3] (see Fig. 2) enables physicists to study the dynamic interaction of kinetic effects that cause a buildup of the edge pedestal in plasma density and temperature profiles and large bootstrap currents with so-called ELMs that may limit pedestal growth and tokamak reactor performance. The overall workflow is implemented using the Kepler [4] workflow management system.

In this workflow, the gyrokinetic PIC edge simulation XGC0 [5] calculates the kinetic edge pedestal buildup. At start-up, the XGC0 code reads in the magnetic equilibrium data (g-eqdsk) that describes the simulated fusion device and the physical experiment, and it uses this data to produce and evolve a plasma profile. At the end of a simulation step, XGC0 writes the plasma profile data (m3d.in), which the MHD code M3D-OMP [6] then uses to update the equilibrium data based on the plasma density, temperature and bootstrap current. XGC0 reads in the updated g-eqdsk data to maintain self-consistency and to continue to evolve plasma particles. The linear MHD stability of the updated equilibrium data is verified by the ideal MHD code, ELITE [7], which executes a parameter sweep over toroidal mode numbers using the plasma density profile data file p-eqdsk, and the magnetic equilibrium...
data file g-eqdsk. If an ELITE step finds an unstable mode, the XGC0 code is stopped and the parallel M3D-MPP code starts a nonlinear simulation of the ELM, which eventually recovers a modified equilibrium data that can be used to start a new XGC0 run and restart the cycle.

Note that the XGC0 and M3D-OMP simulation codes are loosely coupled, e.g., XGC0 may produce plasma profiles more often than M3D-OMP produces equilibrium updates. XGC0 can continue to evolve plasma particles using the current version of the equilibrium data and does not have to wait on M3D-OMP. This is valid because the magnetic equilibrium is not evolving rapidly.

B. Memory-to-memory XGC0–M3D Coupling

We use the kinetic XGC0–M3D-OMP code coupling in the Full-ELM workflow as a case study to illustrate and evaluate the memory-to-memory coupling framework. The framework complements the scientific workflow and manages the communication schedules and data exchanges between the XGC0 and M3D-OMP codes. Other interactions, such as coordination and data exchange with the ELITE code and the visualization and diagnostic modules, are handled by the Kepler workflow system. The overall approach is illustrated in Fig. 3. Client components are integrated with each of the coupled codes, i.e., XGC0 and M3D-OMP, and server components run on a small set of staging nodes. Note that the coupling is two-way, i.e., XGC0 produces and shares plasma profile data, which is used by M3D-OMP, and M3D-OMP in turn produces and shares equilibrium data, which is used by XGC0 in subsequent steps.

The first round of coupling starts with an initial setup step by M3D-OMP, which inserts two copies of the g-eqdsk data object. This object is used as a starting point and is common for the two codes. To compute a new equilibrium, M3D-OMP requests a m3d.in data object from the server component. As this object is not currently available, M3D-OMP blocks. When XGC0 starts, it requests the g-eqdsk data object, and computes and inserts a new version of the m3d.in data object. M3D-OMP can now resume execution and continue to compute a new magnetic equilibrium. When M3D-OMP achieves the desired solution accuracy, it creates and inserts a new g-eqdsk object. Then it starts a new iteration step and retrieves a new m3d.in data object. XGC0 also starts a new iteration step and retrieves a new g-eqdsk data object.

The two simulation codes can now progress independently and at different speeds. For example, a M3D-OMP step for computing the equilibrium with the desired accuracy takes longer than a XGC0 step for evolving the plasma profile. As a result, XGC0 can create and insert multiple m3d.in data objects during a single M3D-OMP step, and M3D-OMP may choose to retrieve one or more of these, e.g., it may retrieve only the most recent version. The coupling scenario is more complex as XGC0 also periodically creates particle profiles for the visualization module, which can make some XGC0 steps longer. During these longer XGC0 steps, M3D-OMP can create and insert more than one g-eqdsk data objects, and again, XGC0 may decide to retrieve only the latest version.

The memory-to-memory coupling is asynchronous and flexible, and it allows the simulation codes to create and share multiple versions and multiple copies of data objects. For example, the g-eqdsk data object has multiple copies that are used by both XGC0 and M3D-OMP, while the m3d.in object is used only by M3D-OMP. The number of coupling rounds and synchronizations are driven by the applications at runtime and are not pre-orchestrated.

V. EXPERIMENTAL EVALUATION

We evaluate the performance and scalability of the memory-to-memory coupling framework using the XGC0–M3D-OMP coupling scenario described in Section IV-B above, on the Jaguar Cray XT5 system at Oak Ridge National Laboratory. Our evaluation is from three perspectives: (1) framework scalability with the number of nodes, (2) framework scalability with the size of the data being shared, and (3) framework behavior with the scientific codes. For the framework scalability experiments, we use synthetic application codes that emulate the XGC0–M3D-OMP coupling behavior and enable us to vary parameters related to data exchange. Finally we evaluate the ability of the sharing framework to satisfy coupling requirements using the real application codes.

A. Evaluating Node Scalability

We conducted three experiments to evaluate the coupling framework scalability with the number of application nodes. In these experiments, we (i) increased the number of nodes inserting and retrieving data and the number of staging nodes running the server component, (ii) increased only the number of nodes inserting data, while keeping the number of nodes retrieving data and the number of staging nodes constant, and (iii) increased the number of nodes inserting and retrieving data, while keeping the number of staging nodes constant. The size of the data objects being inserted or retrieved was fixed at 1MB, and for each experiment all inserted data objects were retrieved irrespective of the number of retrieving nodes. For example, with 16 inserting and 4 retrieving nodes, 16 data objects were inserted and 16 data objects were retrieved in each iteration. In these experiments, we used 100 coupling iterations in which data objects were inserted and retrieved by each of the corresponding nodes. The average insert and
The results of the first experiment in this set are plotted in Fig. 4. In this experiment the number of nodes inserting data objects was increased from 16 to 128, and the number of nodes retrieving data objects as well as the number of staging nodes was increased from 4 to 32. This is a weak scaling experiment and the results show that, as expected, the time for insert and retrieve operations remains approximately constant as the number of nodes increases. The small variations between runs seen on the graph, which are of the order of a few milliseconds, are due to the fact that resources such as the communication links are shared.

In the second experiment the number of nodes inserting data objects was increased from 16 to 128, but the number of nodes retrieving data objects and the number of staging nodes was fixed at 4. The results of the second experiment are plotted in Fig. 5. The results show a very small increase in both, the insert and retrieve times as the number of inserting nodes increase. This is because we are inserting more data objects, while the number of staging and retrieving nodes is fixed. The sharing framework can handle all the requests with a very small overhead on the application side because the size of the data being shared is small. The difference between insert and retrieve times is due to the contention at the staging nodes running the server component. While we do balance load across the server component nodes, small imbalances may exist, which also contribute to this variation.

In the third experiment the number of nodes inserting and retrieving data objects was increased from 16 to 128 and from 4 to 64 respectively, but the number of staging nodes was fixed at 4. The results of the third experiment are plotted in Fig. 6. The results show a minimal increase in the insert and retrieve times (in the order of milliseconds) with the number of nodes inserting and retrieving data objects. This increase is because a larger number of data objects are exchanged while the number of nodes running the server component is fixed.

The goal of the scalability experiments was to analyze and understand the behavior and performance of the coupling framework with an increasing number of application nodes as required in real scientific workflows. We are working on experiments evaluating scalability to larger number of processors and data sizes. We don’t anticipate any limitation in the design of the framework itself and expect the scalability will be limited by available resources such as number of compute nodes, bandwidth of communication links, and the amount of memory at each node.

B. Evaluating Data Scalability

To evaluate the performance of the coupling framework with increasing data object size, we scaled the size of a data object from 1MB to 128MB. In this experiment, we fixed the number of nodes inserting and retrieving data objects and running the server component to 16, 4 and 4 respectively. The results are
plotted in Fig. 7 and represent the average insert and retrieve times over 100 iterations and across the number of inserting or retrieving nodes.

As shown in the plots, the difference between insert and retrieve times becomes noticeable for larger object data sizes. This difference is due to the implementation for the retrieve operation. A node retrieving a data object has to wait for the entire object to be transferred before it can access it. This is due to two main reasons. First, is the semantics of RDMA operations, where an application is notified of a data transfer completion and it can access the content of the corresponding memory buffers only at the end of the transfer, otherwise the object may be in an inconsistent state, e.g., partially transferred. Second, the structure or internal representation of a data object is not known to the server component. As a result, it can only access and transfer entire objects. The retrieve transfer time can be minimized by asynchronously issuing a retrieve call in advance, before the data object is needed.

In this experiment we also compared the data object insert/retrieve times using the memory-to-memory coupling framework with the file write times that would be needed in a file-based coupling workflow. In case of the file-based access experiments, we used the same setup, i.e., 16 nodes producing and writing data objects to a parallel shared network file system [8] and 4 nodes reading the data objects. The results plotted in Fig. 7 are averages over 100 iterations and across the nodes writing/reading. The plot only shows the file write time because the file reads did not finish in the time allocated for the job (i.e., they were significantly larger). The results clearly show that the memory-to-memory coupling framework performs better, especially for large data object sizes. Network file systems amortize network transfers and disk latencies costs by buffering data at the client and then transferring larger buffers, e.g., the write behind policy. However, with larger data sizes, the file system has to flush its buffers more often and the latencies are reflected in the write time. While file writes may be further optimized for an underlying parallel file system, memory-to-memory sharing has the advantage that (1) no disk latencies are involved and (2) the data transfers are asynchronous and the transfer latencies are hidden by overlapping them with computations. We expect that the difference between file-based and memory-to-memory coupling will increase even further as the number of nodes writing and reading increases, primarily due to contention at the filesystem level.

C. Evaluation with Application Codes

Applications that collaborate at runtime have different coordination and data sharing requirements, and we evaluate the flexibility and behavior of our proposed data sharing framework in meeting these requirements using a real coupled fusion simulation scenario and the XGC0 and M3D-OMP production fusion codes. The requirements expressed by these applications are very different, e.g., they run on different resources, share different amounts of data and progress at different speeds. M3D-OMP is used as a helper application for the XGC0 code. It needs new profile data to progress and it has to finish its computations during a timestep of XGC0 execution. As M3D-OMP computations are faster, the code can process and insert multiple equilibrium data objects during one XGC0 timestep, and the XGC0 application must retrieve the latest object at the start of a new execution timestep. Note that the data sharing requirements of these codes are driven exclusively by the physics computations performed by each code.

In the coupling scenario experiment, the XGC0 application ran on 128 nodes, the M3D-OMP application ran on 1 node and the sharing framework ran on 1 node as well. We experimented with four distinct execution setups. In each of these 4 cases, XGC0 ran on the Jaguar compute nodes and uses Portals to communicate with the sharing framework. The M3D-OMP application and sharing framework ran on one of the Jaguar compute node, login node, or a remote cluster node (i.e., Ewok), and used either Portals or TCP for communication to generate the four experimental setups as described below.

First, all three components ran on the compute nodes of the
Jaguar machine and used Portals for communication. Second, the sharing framework and the M3D-OMP application ran on distinct Jaguar login nodes and used Portals for communication. Third, the sharing framework and the M3D-OMP application ran on distinct Jaguar login nodes and used TCP for communication. Finally, the sharing framework ran on a Jaguar login node, the M3D-OMP application ran on the separate Ewok cluster and used TCP for communication. The goal of these experiments is to demonstrate that the sharing framework can couple parallel applications that run on the same machine as well as on distributed resources.

The results of this coupling scenario are presented in Fig. 8 for the XGC0 application and in Fig. 9 for the M3D-OMP application. The labels on the X axis represent the resource that hosted the M3D-OMP application and the type of communication with the sharing framework corresponding to the setups described above.

Due to the requirements of the computation, the XGC0 application inserts the plasma profile object using the application head node and retrieves the equilibrium object using all the nodes. The results show the average time over the number of application iterations and the number of nodes, and the variance across application nodes. The size of the object inserted is smaller than the size of the object retrieved and as the insert operation is asynchronous, the insert time is very close to zero and as a result, does not show in the graph. The retrieve times in the four cases are comparable. The slightly higher value for the “Login/Portals” case is due to the load on the login node, and because the equilibrium data object may not be available right away at the start of a new computation timestep.

In case of the M3D-OMP application, the insert times are also comparable for the four cases. In the “Login/TCP” and “Ewok/TCP” cases the insert operation is synchronous, and this is reflected in the graph. The retrieve time has a higher value, ~12sec, and this is due to application interaction requirements, i.e., M3D-OMP can only continue when new data becomes available. The coordination between the two applications is implicit through the sharing framework, and M3D-OMP has to block and wait for XGC0 to produce new data.

VI. RELATED WORK

Runtime cooperation and data exchanges between coupled simulation processes has been supported using two key approaches (1) messaging frameworks and (2) files.

Messaging-based approaches (e.g., MPI) are optimized for high performance; however, they are less flexible and require simulations to be tightly coupled, the coupling patterns predefined, and the coupled codes be computationally balanced. The file-based approaches often rely on specialized shared file systems (e.g., NFS, Lustre, GPFS) or use workflow coordination frameworks, such as Kepler [4] to move files between applications. They are more flexible and support loose coupling, but they introduce overheads due to the file systems.

Many existing frameworks have focused on addressing the MxN data redistribution problem using different approaches. Projects such as MCT [9] and PAWS [10] used the Common Component Architecture (CCA) to encapsulate the shared data into components. PRMI [11] and related projects use parallel remote method invocation as a programming abstraction for data sharing between heterogeneous applications.

The Distributed Data Broker (DDB) project [12] also targets the data sharing and code coupling problem, but with a different approach. It uses a central resource, i.e., a broker, to collect global data distribution information from the coupled applications, and to compute the communication schedules. Data sharing between applications is orchestrated using the producer-consumer pattern, where producers use the pre-computed communication schedules to send data to consumers. While this is an elegant distributed solution, it assumes pre-defined and static coupling behaviors, imposes tight synchronization requirements, and implicitly assumes that the end applications can directly communicate with each other.

The Seine [13], [14] project provides a transparent data-sharing framework based on associative shared spaces. It provides a simple API for data sharing between simulations with different data decompositions. The Seine framework is specialized for sharing homogeneous data types such as arrays and matrices of doubles. The framework presented in this paper is more flexible and can handle heterogeneous data types.

The DDSS [15] data-sharing framework is conceptually similar to this work. It extends the capabilities of standard libraries such as SHMEM to allow interactions between distinct enterprise applications. In DDSS, the interacting applications have to explicitly program and manage the virtual shared memory for data exchange, e.g., allocate common pages, synchronize accesses for reading and writing, and release reserved resources when they are no longer needed. By contrast, our framework manages communication schedules and memory resources automatically to allow asynchronous and decoupled application data exchanges using high-level abstractions.
The memory-to-memory coupling framework presented in this paper differs from these systems in that it focuses on abstractions for flexible code coupling at the application level, while at the same time providing asynchronous, high-throughput and low-latency data exchange that exploits the advanced communication capabilities of high-performance computing platforms.

VII. CONCLUSIONS

In this paper, we presented the design and implementation of a memory-to-memory coupling framework for interacting scientific simulations on high-performance parallel computing platforms. The framework is driven by the coupling requirements of the CPES prototype FSP center, and provides simple coupling abstractions as well as efficient asynchronous memory-to-memory data transport mechanisms that complement existing parallel programming systems and data sharing frameworks. It enables flexible coupling behaviors and supports dynamic coupling between heterogeneous simulation processes that may progress at different speeds, without enforcing any synchronization constraints.

The evaluation demonstrates the framework performance and scalability in terms of number of nodes and data sizes, and the ability to couple distinct real application codes running on parallel or distributed resources. The evaluation also demonstrates the benefits, in terms of performance and flexibility, of memory-to-memory coupling as compared to file-based coupling in a real simulation scenario.

ACKNOWLEDGMENTS

The research presented in this paper is supported in part by National Science Foundation via grants numbers IIP 0758566, CCF-0833039, DMS-0835436, CNS 0426354, IIS 0430826, and CNS 0723594, by Department of Energy via the grant number DE-FG02-06ER54857, by a grant from UT Battelle, and by an IBM Faculty Award, and was conducted as part of the NSF Center for Autonomic Computing at Rutgers University.

REFERENCES