PROGRAMMING AND RUNTIME SUPPORT FOR ENABLING DATA-INTENSIVE COUPLED SCIENTIFIC SIMULATION WORKFLOWS

By

FAN ZHANG

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Electrical and Computer Engineering

Written under the direction of

Manish Parashar

And approved by

New Brunswick, New Jersey

May, 2015

ABSTRACT OF THE DISSERTATION

Programming and Runtime Support for Enabling Data-intensive Coupled Scientific Simulation Workflows

By FAN ZHANG

Dissertation Director:

Manish Parashar

Emerging coupled scientific simulation workflows are composed of multiple component applications that interact and exchange data at runtime. Coupled simulation workflow enables multi-physics multi-model code coupling and online data analysis, which has the potential to provide high-fidelity modeling and accelerate the simulation data to insight process.

However, running coupled simulation workflows on extreme-scale computing systems presents several challenges. First, most workflow component applications are originally developed as programs that execute independently. Composing a workflow requires the programming support to glue the component applications, orchestrate their executions and express data exchange. Second, simulation workflow requires extracting and moving data between coupled applications. As the data volumes and generate rates keep growing, the traditional disk I/O based data movement approach becomes cost prohibitive and workflow requires more scalable and efficient approach to support the data movement. Third, the cost of moving large volume of data over system interconnection network becomes dominating and significantly impacts the workflow execution time. Minimize the amount of network data movement and localize data transfers in the network topology is critical for reducing such cost. To achieve this, workflow task placement should exploit data locality to the extent possible and move computation closer to data.

This thesis addresses these challenges related to workflow composition, data management and task placement, and makes the following contributions: (1) This thesis presents DIMES data management framework to support memory-to-memory data movement between coupled applications. DIMES co-locates in-memory staging on application compute nodes to store data that needs to be shared or exchanged, and enables accessing the data through array-based query interface. (2) This thesis presents CoDS task execution framework to support workflow composition and execution, which implements the task execution programming interface for composing customized workflow and orchestrating the execution of component applications. (3) This thesis presents communication- and topology-aware task mapping, which implements a holistic approach to map workflow communication graph onto physical network topology. The method effectively reduces the total size of network data movement and reduces the workflow communication time. The research concepts and software prototypes have been evaluated using real application workflows on extreme-scale computing systems.

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Manish Parashar, for his guidance and support throughout my doctoral study and research. I am grateful for his advice, encouragement, patience and cheerfulness during my years at Rutgers.

I would like to thank the members of my committee, Dr. Ivan Marsic, Dr. Ivan Rodero, Dr. Deborah Silver and Dr. Scott Klasky, for taking time off their busy schedule to read and review my thesis.

I would like to give special thanks to the DataSpaces team, Hoang Bui, Tong Jin, Qian Sun and Melissa Romanus, with whom I spent the best part of my years at Rutgers. Much of this work would not have been possible without their contributions.

I thank Dr. Scott Klasky from Oak Ridge National Laboratory for his collaboration and the access to HPC resources. Thanks to Norbert Podhorszki, Jong Youl Choi, Hasan Abbasi at ORNL ADIOS team for their collaborations, valuable discussions and co-authoring research papers.

I would also like to thank my friends and colleagues at Rutgers Discovery Informatics Institute (RDI^2) for creating a collaborative, productive and motivating work environment.

I owe enormous thanks to my parents, Jianxun Zhang and Zhenmei Chen, who have been a persistent source of love and encouragement for me. Lastly, I express my gratitude to dear saints in the local church, for their love and support.

Dedication

To my wife Yahong.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

With the fast growing computational power of supercomputers, scientific simulation is able to realize finer granularity and plays an important role in driving cutting edge scientific research. Moreover, scientific simulations increasingly formulate as coupled workflows that consist of multiple interacting applications, which enables code coupling and online data analysis, and has the potential to achieve high-fidelity modeling and accelerate the data to insight process.

This dissertation addresses the challenges associated with programming and runtime support for enabling coupled simulation workflows on extreme-scale computing systems. In this introductory chapter, it starts with brief descriptions on the background and research problems, then presents an overview of the thesis research and contributions. This chapter concludes with the outline of the dissertation.

1.1 Background

Emerging coupled simulation workflows are composed of multiple applications that interact and exchange data at runtime. Simulation workflows running at extreme scale have the potential to achieve higher accuracy and accelerate the data to insight process. Multi-physics multi-model simulation workflow simulates different aspects of the phenomena being modeled by coupling multiple physical models. For example, in the Community Earth System Model (CESM) [28] application workflow, separate simulations are coupled to model the interaction of the earth's ocean, atmosphere, land surface and sea ice. Meanwhile, online data analytics workflow supports analyzing raw simulation data while it is being generated. For example, online analytics workflow for combustion simulation S3D [27] extracts and streams simulation data to a number of analysis operations, e.g., visualization, descriptive statistics, which execute concurrently and in parallel with the simulation.

However, running coupled simulation workflows on extreme-scale computing system is non-trivial. Simulation workflows are generating large volumes of data that needs to be dumped to storage system, shared with another coupled application, or analyzed and processed in a timely manner to extract scientific insights. As the volumes and generation rates of the data grow, the costs associated with extracting data from simulation and transporting it for coupling and analysis have become the dominating overheads, which makes the traditional disk I/O based data interaction approach cost prohibitive and often infeasible.

In order to mitigate the increasing performance gap between computation and parallel I/O, the *data staging* approach has been proposed to accelerate I/O operations, support memory-to-memory data coupling, and enable online simulation-time data analysis and processing. In this approach, data staging software builds a scalable data management layer on the extreme-scale machine, and utilizes DRAM, Non-volatile Memory (NVRAM) or Solid State Drive (SSD) for staging the simulation data. Recent research work in data staging has focused on the following directions.

I/O forwarding and acceleration. I/O forwarding technique has been used to transparently offload expensive I/O operations to dedicated data staging nodes or I/O nodes. This technique enables asynchronous I/O and reduces the impact of I/O on simulation. Examples include DataStager [10], GLEAN [63], ADIOS [53], Nessie [57], IOFSL [13]. Similarly, burst buffer has been proposed to absorb bursty application I/O behaviors by integrating NVRAM or SSD based buffers into the compute nodes or I/O nodes on extreme-scale systems. Examples include Tianhe-2 [68], NESRC Cori [6] and ORNL Summit [8] supercomputers.

Runtime data coupling enables efficient memory-to-memory data sharing between applications that are part of a coupled workflow. Data staging software, such as DataSpaces [35], Flexpath [29], H5FDdsm [19], builds a scalable data management layer for staging of simulation data that needs to be shared or exchanged, and utilizes the high-performance network interconnect for data movement.

Online data analysis and processing. Data staging software can be used to build online data analysis and processing systems that analyze the simulation data while it is being generated. Analysis operations can be placed *in-situ* on the same compute nodes that run the simulation code, such as in Darmaris/Viz [37], FP [52], GoldRush [74], or *in-transit* on a set of separate and dedicated staging nodes, such as in ActiveSpace [34], PreDatA [73]. Several recent research efforts, e.g., JITStager [9], FlexIO [75], DataSpaces [16, 46] also provides the flexibility of supporting both in-situ and in-transit analysis placement.

Recent data staging research brings opportunities for improving the performance of parallel I/O, data sharing and analysis in a coupled simulation workflow. However, enabling workflow execution on extreme-scale systems still presents programming and runtime challenges in the following aspects: (1) Workflow composition - compose a workflow by coupling the component applications; (2) Data management - manage the data sharing and exchange between coupled applications; (3) Task placement - place the application processes onto the physical compute nodes and processor cores. These challenges are the focus of this thesis and are described in more details in the next section.

1.2 Research challenges

Coupled simulation workflow builds on component applications that are originally developed as programs that often execute independently, without considering the need for coupling. As a result, composing a workflow using existing component applications requires programming support. First, **workflow composition requires the programming support for orchestrating the application executions**. This allows programmers to express the control flow for the workflow execution, specify data dependencies between applications. Second, **workflow composition requires high-level data abstraction that allows programmers to effectively express the data sharing and exchange between the coupled applications**.

Coupled simulation workflow requires extracting data from one simulation, and transferring and redistributing the data to another simulation or analysis. Traditionally, workflow applications share and exchange data using distributed file systems, such as Networked File System (NFS) [61], Parallel Virtual File System (PVFS) [23], General Parallel File System (GPFS) [62], Lustre [5]. The file-based disk I/O approach enables applications to use common data access interfaces (e.g., POSIX I/O) and a wide variety of scientific data formats (e.g., HDF5 [4], NetCDF [7]), which brings portability and flexibility to the software development. But the increasing performance gap between computation and disk I/O introduces significant overhead and limits the data exchange performance. As the volumes and generation rates of the data grow, the traditional disk I/O based coupling approach becomes cost prohibitive and often infeasible.

Memory-based coupling approaches have been developed to address the increasing performance gap between computation and disk I/O. *Direct data movement* approaches, such as Model Coupling Toolkit [49], FlexPath [29], enable memory-to-memory data redistribution directly between coupled applications. Direct data movement reduces the latency by avoiding extra data transfers. However, existing implementations of direct data movement only support sharing data between concurrently running applications, because the availability of data is dependent on the presence of data producer applications. *Staging-based data movement* approach, such as DataSpaces [35], H5FDdsm [19], uses dedicated staging servers to support data sharing and exchange between the coupled applications. The more persistent storage space on staging servers is capable of supporting data exchange for both concurrently and sequentially coupled applications. But staging-based data movement introduces performance overhead as it transfers data twice, from application to the staging servers and then to the application. Moreover, this approach requires allocating additional server resource to buffer the data. We need a **data management mechanism that can support memory-to-memory data sharing for different coupling patterns, provide high-performance data movement, and minimize the use of additional resource.**

Even with the memory-based approach, moving large volume of data over system interconnection network can still impact the performance of workflow execution. Workflow applications are often tightly coupled, and the data exchange is performed frequently. The cost (latency and energy) associated with data exchanges in these cases can dominate. Minimizing the amount of data movement over network, and localizing data transfers within the network topology is critical for reducing such costs. Task placement determines the mapping of workflow application processes to physical processor cores, and plays a significant role in minimizing and localizing data movement.

However, effective task placement on extreme-scale computing systems is challenging. System architecture of the largest supercomputers becomes highly hierarchical, with increased core count per compute node and increased number of compute nodes that are interconnected by complex network topologies, e.g., 3D torus, Fat tree, Dragonfly. On-node data movement is more efficient than off-node data movement that requires transferring data over the interconnection network. Data movement between compute nodes that are nearby in the network topology is more efficient than between compute nodes that are long-hop away. As a result, **workflow task placement should exploit data locality to the extent possible, increase the size of on-node data exchange by moving computation closer to data, and map the workflow communications onto the interconnected compute nodes in a topology-aware manner.**

1.3 Overview of thesis research

The overall goal of this thesis is to address the research challenges related to workflow composition, data management and task placement for coupled simulation workflows, which are described above. This section presents an overview of thesis research.

This thesis presents the DIMES data management framework, which implements the programming interface and runtime mechanism for sharing data between coupled applications. DIMES provides a shared array data model, and implements array-based spatial query interface to enable applications to access data of interest. In scientific simulations, multidimensional arrays are commonly used as the main data structure to store the raw simulation data. Data exchange between applications typically requires moving and redistributing arrays from one simulation to the other. The DIMES array data model and query interface can meet the programming requirement of most coupled scientific simulation workflows.

DIMES co-locates data staging with application execution on the same set of compute nodes, and utilizes node-local storage resource, e.g., DRAM, to cache application data that needs to be shared, exchanged or accessed. Co-located data staging provides low-latency high-throughput write performance and significantly reduces the volume of data movement over network as compared to *staging-based data movement* approaches. In addition, DIMES implements location aware data movement. Depending on the data locations, e.g., local or remote memory, DIMES dynamically selects the appropriate transport mechanism to support high-performance data transfer. For example, DIMES uses hardware-supported RDMA network operation for fetching data resides on remote compute nodes, and uses direct memory access to fetch data in node-local shared memory segment.

This thesis also presents the CoDS task execution framework, which implements the programming interface and runtime mechanism for task execution and placement. CoDS extends the Directed Acyclic Graph (DAG) task graph abstraction used by traditional scientific workflow systems to represent coupled simulation workflows. Its programming environment is exposed as a library to existing languages, and implements task execution APIs to compose a workflow. CoDS workflow consists of a driver program and a number of task programs. Each task program represents a component application, and the driver program couple the component applications and compose the actual workflow by combining sequential, concurrent or iterative execution of task programs.

CoDS implements locality-aware task placement. It attempts to place task execution on compute nodes that contain the largest portion of its input data. In particular, CoDS allows programmers to express locality preferences for a task by providing *data hints*, e.g., name and spatial region of the array data that to be accessed by the task. In addition, CoDS allows users to programmatically customize and control the task placement, according to the specific needs of the targeted workflow. Programmers can divide the allocated computational resource into functional partitions, and explicitly express placement affinity by providing *location hint*, e.g., the preferred functional partition for the task execution. For example, online data analytics workflow requires the flexibility [72, 16, 75] to place analysis either "in-situ" on the same compute node that runs the simulation code, or "in-transit" on a dedicated set of compute nodes. CoDS enables programmers to construct such multi-level functional partitions, e.g., "simulation", "in-situ", "in-transit", and flexibly control placement.

Finally, this thesis presents communication- and topology-aware task mapping, which implements a holistic approach for mapping workflow communications onto a physical network topology, with the goals of reducing the communication costs. The key underlying idea is to *localize* communication at two levels. First, the task mapping improves data locality by placing intensively communicating processes onto the same multi-core compute node, in order to reduce the volume of data movement over the network fabric. Second, the task mapping improves data locality by placing communicating processes onto physically "close" compute nodes in the network topology, in order to reduce the number of hops for network-based data transfers. While existing research [69, 11, 18, 66] has focused on mapping frequently communicating processes of a single application, mapping coupled simulation workflows consisting of multiple interacting applications has not been studied. Our method targets coupled simulation workflow, and aims at reducing the cost caused by both intra-application and inter-application data movement.

1.4 Contributions

This thesis makes the following contributions.

- Design and implementation of DIMES, which provides programming and runtime support for data sharing and exchange between coupled applications. DIMES enables distributed in-memory data staging, and high-performance memory-to-memory data movement.

- Design and implementation of CoDS, which provides programming and runtime support for workflow task execution and placement. CoDS supports locality-aware and flexible task placement, by utilizing programmer-provided placement hints information, e.g., *data hints* for expressing locality preference, *location hints* specifying execution location preference, etc..

- Design and implementation of communication- and topology-aware task mapping for coupled simulation workflows, which effectively reduces the total size of data movement over network fabric and reduces the *hop-bytes* for network-based data transfers.

- Integration of the prototype implementations with several real-world coupled simulation workflows in combustion and plasma fusion.

1.5 Thesis outline

The rest of this thesis is organized as follows.

Chapter 2 presents motivating and representative workflow examples, and summarizes the programming and runtime requirements.

Chapter 3 presents a high-level overview of the technical approaches for realizing the programming and runtime support.

Chapter 4 presents the design, implementation, and evaluation for DIMES.

Chapter 5 presents the design, implementation, and evaluation for CoDS.

Chapter 6 presents communication- and topology-aware task mapping for optimizing the communication performance of coupled simulation workflow.

Chapter 7 summarizes the research work, presents concluding remarks and directions for future work.

Chapter 2

Motivating applications and requirements

This research is largely driven by the experiences of dealing with the problems of data management and task placement that arise in real world workflow applications. This chapter presents representative workflow examples, describes and summarizes the key programming and runtime requirements for supporting coupled simulation workflows on extreme-scale computing systems.

2.1 Motivating coupled simulation workflow scenarios

2.1.1 Online data analytics workflow for combustion simulations

Workflow description

S3D data analytics workflow couples S3D [27] - a massively parallel turbulent combustion simulation with data analysis operations, in order to enable online analysis of raw simulation data while it is being generated. This online processing approach enables more fine-grained exploration of raw simulation data, which is not feasible with the traditional post-processing approach due to the significant I/O overhead.

Figure 2.1 illustrates the data interactions between coupled applications, as well as the execution logic of applications. At runtime, the workflow periodically extracts and transfers selected chemical species to the analysis operations. In this specific example, S3D is coupled with three data analysis operations including parallel visualization (VIZ), descriptive statistics (STAT), and topological analysis (TOPO). As shown in Figure 2.1, analyses are performed with different frequencies. For example, visualization is executed after every 2 simulation time steps, while descriptive statistics is executed after each simulation time step.

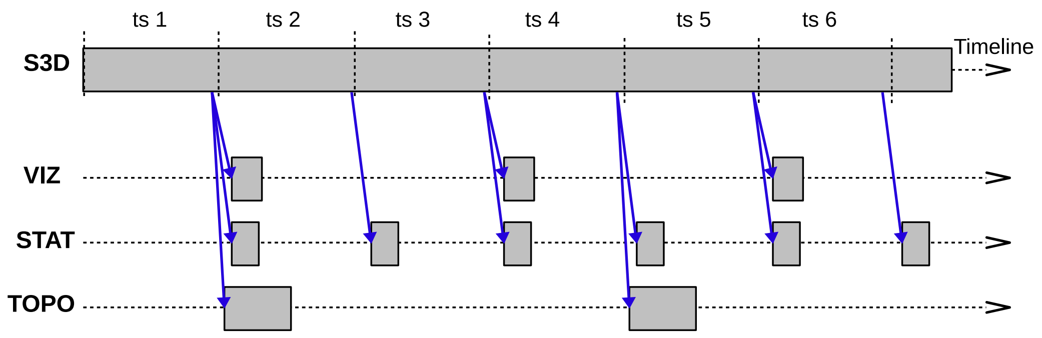


Figure 2.1: Illustration of data interactions between coupled applications and execution logic for the S3D data analytics workflow. Solid arrows denote the data flow between workflow applications.

Requirements

First, the workflow requires timely extraction and redistribution of chemical species from S3D simulation to analysis operations. Overlapping data movement with the computation is desirable, which can reduce the data movement overhead as perceived by the simulation. Second, in order to effectively utilize the computation resource allocated for the workflow, it requires data-driven execution which starts the analysis only when all required input data becomes available. Third, the workflow requires placement flexibility for analysis tasks. For example, the workflow can execute an analysis task in-situ to reduce the amount of data movement over network, if the communication between S3D and the analysis is a dominant cost factor. Similarly, the workflow should also have the ability to execute an analysis task in-transit to minimize the impact on simulation, if the cost of communication between S3D and the analysis is insignificant.

2.1.2 Coupled DNS-LES workflow for combustion simulation

Workflow description

Coupled DNS-LES workflow couples the direct numerical simulation (DNS) solver with the large eddy simulation (LES) solver in order to perform a model assessment for combustion science. The DNS solver provides a high degree of accuracy on fine grids, but the accuracy comes by running the simulation on a large number of processor cores and keeping the simulated timescale small. As a result, it is very expensive to use DNS to simulate longer timescales that are typically the conditions of practical interest. In contrast, LES models can be used with coarser grids to simulate longer timescales with fewer computation resource, but with decreased simulation accuracy. Coupling DNS and LES provides a testbed to enable development and assessment of practical LES models. DNS and LES executes concurrently in lockstep and computes on identical physical domain, but with different grid sizes. At each time step, the base solution field computed by DNS simulation is filtered and transferred to LES simulation.

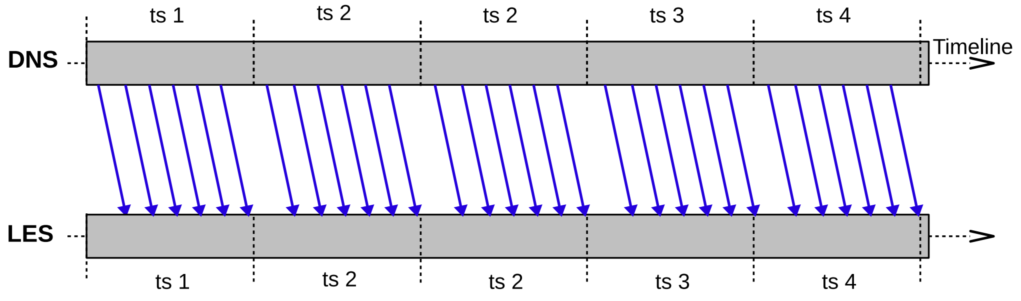


Figure 2.2: Illustration of data interactions between coupled applications and execution logic for the coupled DNS-LES workflow for combustion science. Solid arrows denote the data flow between workflow applications.

Figure 2.2 illustrates the data interactions between coupled applications, as well as the execution logic of applications. In this specific example, DNS is coupled with one instance of LES. Data is transferred from DNS to LES at every sub step, i.e., 6 times in a single time step, and each time step typically requires only a few seconds of wall-clock time. LES can not proceed to its computation phase until it receives all required data fields for current sub step.

Requirements

First, the workflow requires scalable and low-latency data movement for the highly frequent interaction between DNS and LES solvers, in order to support the lockstep execution. Second, as the workflow executes on increasing number of compute nodes and exchanges large volumes of data, the communication cost can becomes dominating. As a result, task placement that takes into consideration both the communication pattern and system network topology is desirable to optimize and reduce the communication cost.

2.1.3 Coupled XGC1-XGCa workflow for plasma fusion simulation

Workflow description

Coupled XGC1-XGCa workflow provides insight into plasma edge physics in magnetic fusion devices, and consists of two axisymmetric kinetic transport codes XGC1 and XGCa [48, 30]. Simulation of plasma edge is a multi-scale problem that involves many disparate time and length scales. Though XGC1 could solve multi-scale problem alone, XGC1 has numerical dissipation and model equation errors, and requires significant computing power when simulating large quantity of particles. In contrast, XGCa uses a coarser mesh and requires much fewer particles, and can be used as an accelerator for XGC1.

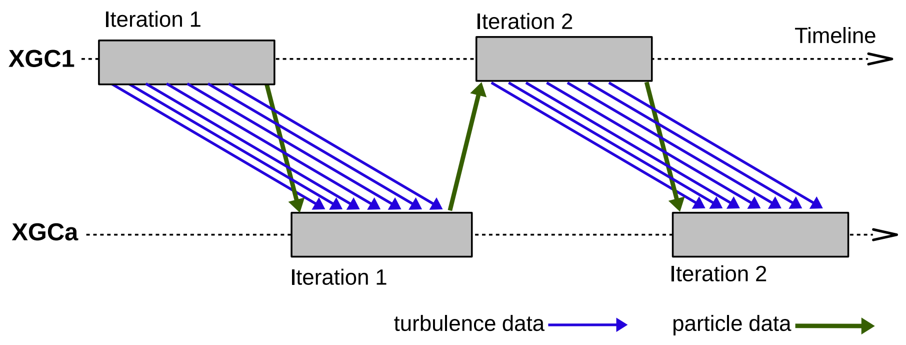


Figure 2.3: Illustration of data interactions between coupled applications and execution logic for the coupled XGC1-XGCa workflow. Solid arrows denote the data flow between coupled applications.

Figure 2.3 illustrates the data interactions between coupled applications, as well as the execution logic of applications. XGC1-XGCa workflow executes for multiple coupling iterations. In each coupling iteration, the workflow first executes XGC1 for n (n is 7 in the illustrated example) time steps to compute turbulence data and particle state. XGC1 needs to write turbulence information generated at each time step. After the plasma has evolved to a quasi-steady turbulent state, the execution is switched to XGCa to evolve the background profiles using the turbulence data from XGC1. XGCa needs to read the turbulence data at each time step in order to proceed its execution.

Requirements

First, data exchange between the sequentially coupled XGC1 and XGCa is decoupled in time. Memory-based coupling requires the ability to cache intermediate data generated by the simulations, and lookup and access the cached data on-demand. Second, as workflow execution is being switched between the two simulations, locality-aware task placement is desirable to execute newly launched simulation processes on compute nodes which have cached the largest portions of the required input data.

2.2 Programming and runtime requirements

2.2.1 Workflow composition

As illustrated by the examples above, coupled simulation workflows are typically composed of multiple applications. However, most component applications are originally developed as independent programs, without considering the need for coupling. As a result, extended programming support is required to compose a workflow and couple the component applications.

First, composing a workflow requires the programming support to express the control flow for the workflow execution, and specify the data dependency between applications (or called tasks). In addition, many coupled simulation workflows require the support for iterative or data-driven execution, which dynamically spawns task execution based on previous computation results. As a result, programming support for workflow composition also requires the ability to express dynamic task execution. Traditional scientific workflow management systems [3, 32, 12] use directed acyclic graph (DAG) based task graph abstraction to represent a workflow, where each graph vertex represents a workflow task and each edge identifies dependency between tasks. However, most existing systems support static DAG that needs to be fully specified before executing the workflow, which can not express dynamic task execution.

Second, composing a workflow requires the programming support to express the data sharing and exchange between the coupled applications that are part of a workflow. Generic low-level parallel programming abstractions, such as message-passing interface (MPI) [42], can be used to implement the data exchange. However, it is nontrivial to program with such low-level communication interface, given the fact that data is distributed over a massive number of parallel processes in each application and the interacting applications may have distinct data distribution types. Partitioned global address space (PGAS) programming languages [22, 26, 56, 25] emerge recently and provide high-level data abstraction, such as shared array, which is more expressive and hides low-level details. However, PGAS shared array data abstraction is language-dependent and only accessible by processes (or threads) in the same application. It currently does not support sharing data between applications of distinct programming models, such as between PGAS and MPI applications. As a result, we need data abstraction and programming interface that is expressive, independent of specific parallel programming languages. This would allow programmers to effectively express data sharing and exchange between heterogeneous coupled applications.

2.2.2 Data management

As illustrated by the examples above, data management for coupled simulation workflow requires staging, transfer and redistribution of a large volume of data between the coupled applications. This section summarizes the requirements for data management.

Data indexing and lookup: Data exchange for coupled simulation workflow requires moving data, such as global array, from one application running on M processes to another application running on N processes, i.e., M×N data redistribution problem. For example, each data consumer application process may read a subarray of the global array data for local computation. Efficient M×N data redistribution requires the ability to index the data generated by the data producer application and support quick data lookup for each read request.

High-performance data movement: Coupled simulation workflow requires efficient and scalable data movement to support coupling at scale. First, the data movement needs to be low latency, and overlap with the application computation, so as to minimize the overhead perceived by the applications. For example, the DNS-LES workflow requires fast memory-to-memory data movement to support the highly frequent data interaction between DNS and LES solvers. Second, the data movement needs to utilize the most suitable data transport mechanism. For example, the data movement may be between processes on the same compute node, or on different compute nodes. It is desirable to adapt the underlying transport mechanism based on data location, e.g., using network data transfers for remote data movement, and shared memory which bypasses network devices for node-local data movement.

2.2.3 Task placement

Task placement determines where the workflow tasks are executed, and the mapping of application processes to physical processor cores, which has significant impact on the data communication performance. This section summarizes the requirements for task placement.

Locality-aware and flexible task placement: Locality-aware task placement that moves computation closer to data is critical for reducing the size of data movement over network. In addition, recent research [72, 16] has shown the need for flexible placement of data analysis tasks in online data analytics workflow, such as in-situ analysis, in-transit analysis or combined in-situ/in-transit analysis. As a result, it requires the ability to flexibly place the analysis tasks, in order to meet the various placement requirements.

Topology-aware task mapping: To effectively reduce the cost of data movement in coupled simulation workflow, task mapping needs to consider both the communication pattern exhibited by the workflow and the network topology of interconnected compute nodes. For example, appropriate placement of heavily communicating application processes onto nearby compute nodes in the network topology can reduce the number of hops for data transfer, and potentially reduce the contention by localizing the communications in the network.

Chapter 3

Overview of technical approaches

Chapter 2 describes the programming and runtime requirements for supporting coupled simulation workflows. This chapter presents an overview of the technical approaches and implementations for realizing the required programming and runtime support. In addition, this chapter describes the foundational technology, i.e., DataSpaces framework, which is used by implementations presented in this thesis.

3.1 Overview of technical approaches and implementations

A high-level overview of the technical approaches for addressing the programming and runtime requirements is presented as follow.

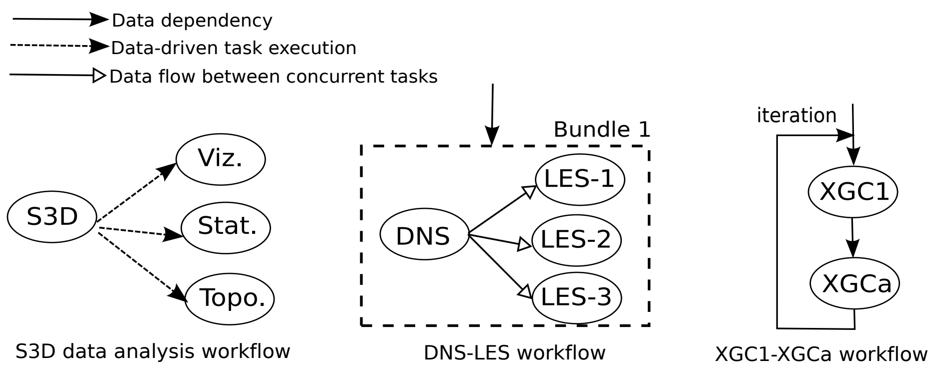


Figure 3.1: Illustration of task graph based representations for the example workflows.

The proposed programming approach for workflow composition uses the **task graph abstraction** to represent a workflow, which is based on the DAG task graph used by traditional scientific workflow management systems [3, 32]. Each vertex in the DAG represents a task, i.e., workflow component application, and each edge represents the data dependency. A task can start after all its parent tasks finish executions. In order to represent coupled simulation workflows, we extend the basic DAG task graph abstraction with the abilities to annotate the data flows between concurrently executing tasks, express iterative and data-driven task execution. The task graph abstraction represents the control flow and data dependency between applications, and captures the inter-application communication. Figure 3.1 illustrates the task graph based representations for the workflows described in Chapter 2.1. A detailed description of the workflow representation is presented in Chapter 5.2. The proposed programming approach is exposed as a task execution library to existing language. Programmer develops a driver program to compose a workflow, which implicitly implements the workflow task graph. A driver program orchestrates the executions of task programs that are part of a workflow, and drives the workflow progression.

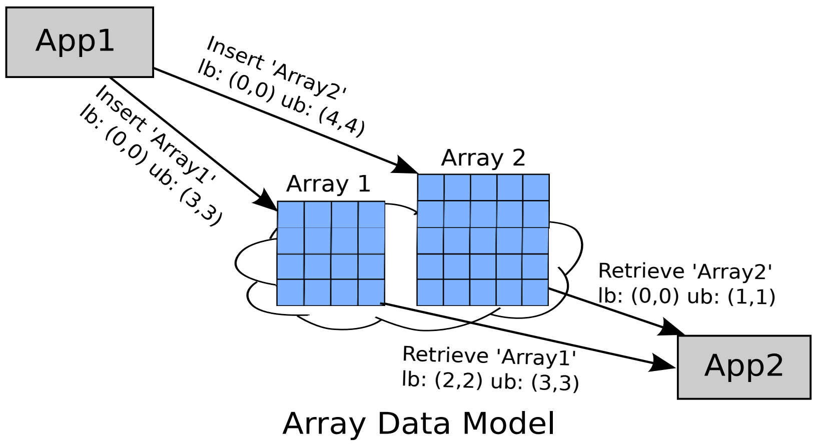


Figure 3.2: Illustration of sharing two 2D arrays through the array-based data model. *lb, ub* denote the bounding box of the inserted/retrieved array region.

The proposed programming approach for workflow composition uses **array-based data model** to express data sharing and exchange. In scientific simulations, a multidimensional array is commonly used as the main data structure to store data, where the global array is decomposed into non-overlapping subarrays and distributed over the simulation processes. Data exchange typically involves moving and redistributing arrays from one simulation to another. Array-based data model presents the shared array abstraction, and enables different applications to access the shared array data using specialized insert/retrieve query interface. Figure 4.2 illustrates the sharing of two 2D arrays between applications using the array-based data model. This higher level data model provides a more natural way for applications to share and exchange array-based scientific data.

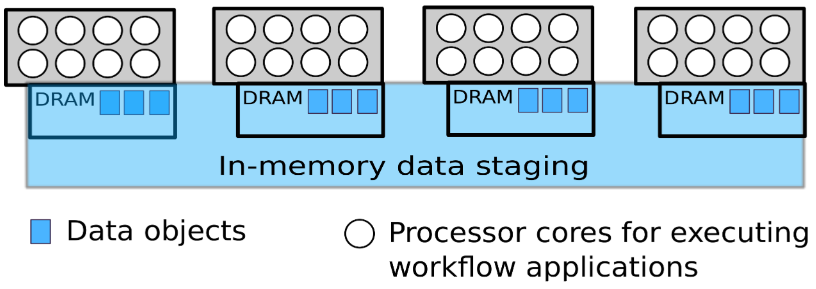


Figure 3.3: Illustration of co-locating in-memory data staging on application compute nodes.

This thesis proposes **co-located in-memory data staging** approach to support run-time data management, and enable memory-based parallel data exchange between coupled applications. Unlike existing data staging approaches that store data on a set of dedicated compute nodes, our approach co-locates the data staging on the same set of compute nodes that execute the workflow applications, and utilizes the node-local storage resource to cache data that needs to be shared or exchanged. Figure 3.3 illustrates the co-located in-memory data staging. This approach caches data in local memory and thus provides low-latency high-throughput write performance, and enables transferring data directly between coupled applications.

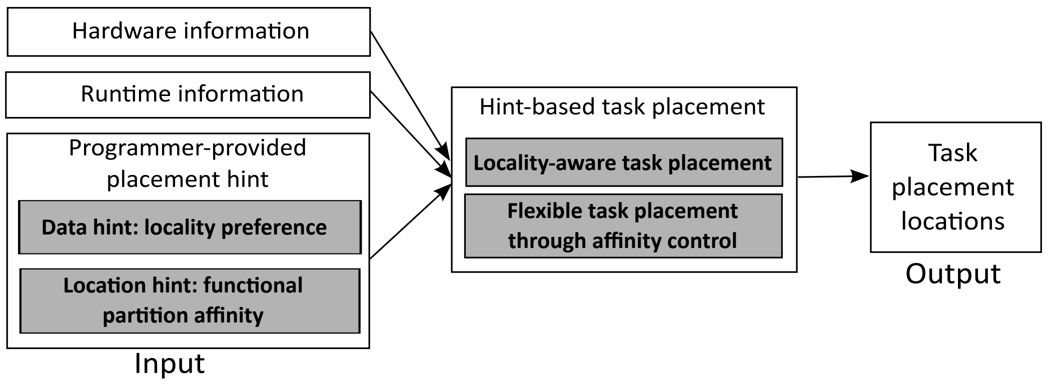

Hardware information
Runtime information
Programmer-provided placement hint
Data hint: locality preference
Location hint: functional partition affinity
Input
Hint-based task placement
Locality-aware task placement
Flexible task placement through affinity control
Task placement locations
Output

Figure 3.4: Conceptual view of the hint-based task placement.

This thesis proposes **hint-based task placement**, which combines the programmer-provided hint information and runtime status to make a better task placement decision. Figure 3.4 illustrates the conceptual model of the approach. Programmers can specify placement hint according to their knowledge and expertise, which provides a piece of insightful information that can be utilized by the runtime system. Programmers can express locality preference hint for a task, e.g., name and spatial region of the task's input array data, which is used by the runtime to perform locality-aware placement and move task computation closer to the required data. In addition, hint-based task placement enables application scientists to functionally partition the computation resource and explicitly express the placement affinity of a task, i.e., the preferred functional partition for the task execution. For example, as described in Chapter 2.2, online data analytics workflow requires placement flexibility for analysis tasks that are coupled with the simulation. In this case, programmers can partition the computation resource to create different placement locations, e.g., “simulation”, “in-situ”, “in-transit”, and control which partition to execute a task by providing the location preference hint.

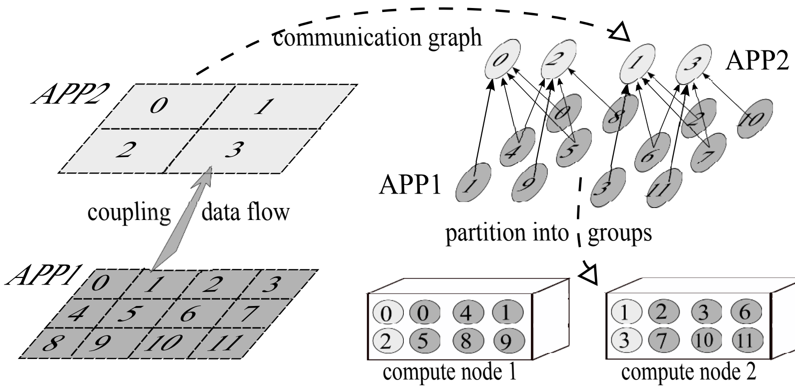


Figure 3.5: Illustration of communication- and topology-aware task mapping.

This thesis proposes a **communication- and topology-aware task mapping** approach to minimize the communication cost in a coupled simulation workflow, by reducing the volume of data transfers over network as well as the number of hops for network data transfers. This approach essentially performs the mapping of workflow application processes to the physical compute nodes interconnected by the network fabric. It analyzes the logical communication graph of a workflow, including both intra-application and inter-application communications, and applies graph partitioning and topology mapping algorithms to map the communication graph onto the network topology graph. Figure 3.5 uses a simple example to illustrate the task mapping. APP1 runs 12 processes and APP2 runs 4 processes, and each compute node has 8 processor cores. In this example, the task mapping reduces the data movement over network by placing heavily communicating processes onto the same multi-core compute node.

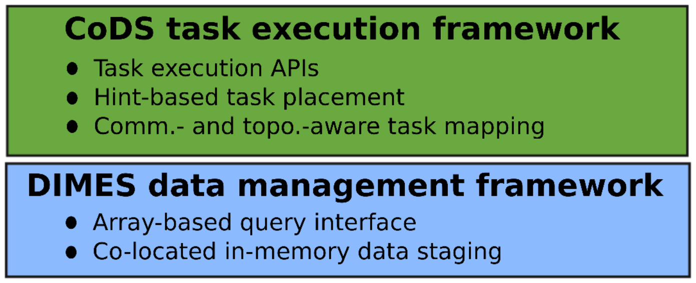

CoDS task execution framework
Task execution APIs
Hint-based task placement
Comm.- and topo.-aware task mapping
DIMES data management framework
Array-based query interface
Co-located in-memory data staging

Figure 3.6: Overview of thesis technical implementations.

Figure 3.6 presents an overview of the implementations, including DIMES data management framework and CoDS task execution framework. DIMES implements the co-located in-memory data staging, and implements the data insert/retrieve query interface based on array data model. Chapter 4 presents technical implementation details for DIMES. CoDS implements the task execution APIs to compose workflows that can be represented using the proposed task graph abstraction, and implements the hint-based task placement. Chapter 5 presents technical details for CoDS. In addition, we present the communication- and topology-aware task mapping approach in Chapter 6.

3.2 Foundational technology - DataSpaces

The current prototype implementation builds on DataSpaces [35], which is a scalable data-sharing framework targeted at current large-scale systems and designed to support dynamic data interaction and coordination between coupled scientific applications. DataSpaces provides a semantically specialized shared space abstraction using a set of staging nodes. This abstraction is derived from the tuple space model [24] and can be associatively accessed by the interacting applications in a coupled simulation workflow. DataSpaces also provides services for asynchronously extracting and indexing data from running applications, and enables this data to be flexibly queried. DataSpaces is built on an RDMA-based asynchronous memory-to-memory data transport layer called DART [36].

DataSpaces has been integrated with and deployed as part of the Adaptive IO System (ADIOS) [54] framework distributed by Oak Ridge National Laboratories. ADIOS is an open source I/O middleware package that has been shown to scale to hundreds of thousands of cores and is being used by a wide range of applications.

Chapter 4

DIMES - a distributed in-memory data staging and coupling framework

4.1 Introduction

Data-intensive coupled simulation workflows require data generated by the data producer application to be transferred and redistributed to the data consumer application. For example, in the data analysis workflow for the S3D combustion simulation, variables such as temperature and velocity needs to be transferred to multiple user-defined analysis operations for online processing. Similarly in the climate modeling workflow that couples different geophysical simulations such as atmosphere, land and sea-ice, boundary data consisting of a large number of data fields needs to be frequently exchanged between the coupled component models.

This chapter presents **DI**stributed **ME**moray **S**pace (DIMES) data management framework to support distributed in-memory staging, and memory-to-memory data sharing and exchange between applications that are part of a workflow. DIMES co-locates data staging with application execution on the same set of compute nodes, and utilizes node-local storage resource (e.g., DRAM in current prototype) to cache and store application data that needs to be shared, exchanged or accessed. In scientific simulations, a multidimensional array is commonly used as the main data structure to store data, where the global array is decomposed into non-overlapping subarrays and distributed over the simulation processes. Data exchange typically involves moving and redistributing arrays from one simulation to another. DIMES provides array data model to user applications, and its array-based query interface enables applications to retrieve data of interest by specifying spatial constraint, e.g. Cartesian bounding box. In addition, DIMES implements data-location aware data movement strategy. Depending on the data locations, e.g., local or remote memory, DIMES dynamically selects the appropriate transport mechanism to support high-performance data movement. For example, DIMES uses hardware-supported RDMA network operation for fetching data resides on remote compute nodes, and uses direct memory access to fetch data in node-local shared memory segment.

Technical contributions of DIMES are summarized as below:

- DIMES provides scalable and efficient inter-application data sharing and exchange for coupled simulation workflow. DIMES implements data-location aware data transfer using high-performance RDMA network operations and on-node shared memory.

- DIMES co-locates data staging with workflow execution. DIMES implements a distributed in-memory object store on the compute nodes that run the workflow, and caches application data in node-local memory. Co-located data staging provides low-latency high-throughput write performance and reduces the volume of data movement over network.

- DIMES supports data exchange for both concurrently and sequentially coupled applications. Existing approaches [49, 75, 29] can only support memory-based data movement between simultaneously executing applications, and does not support data exchange between sequential applications. DIMES has the capability to cache data in on-node persistent shared memory segment, and enables access to data even after the data producer application finishes its execution.

- DIMES is integrated into Adaptive IO System (ADIOS) I/O middleware [54], and enables a large number of existing applications to benefit from in-memory coupling. Applications using ADIOS APIs can switch from file-based to memory-based data sharing, by changing the transport method in ADIOS configuration XML file. This brings productivity and portability to application development.

- Hybrid data staging is supported by combing in-memory staging on both application compute nodes and dedicated staging nodes. DIMES builds on and derives from our previous work on DataSpaces, which caches data on a dedicated set of compute nodes, and can co-exist with DataSpaces. As a result, coupled simulation workflows have access to both local data staging and remote data staging (on DataSpaces servers).

The remainder of the chapter is organized as follows. Section 4.2 describes DIMES data abstractions. Section 4.3 presents DIMES system architecture. Section 4.4 presents the design and implementation. Section 4.5 presents the programming interface and examples. Section 4.6 presents evaluation results. Section 4.7 presents related research work and Section 4.8 summarizes this chapter.

4.2 Coordination and data model

Workflow applications can express data exchange using two types of data abstractions that are provided by DIMES, including shared space model and array data model.

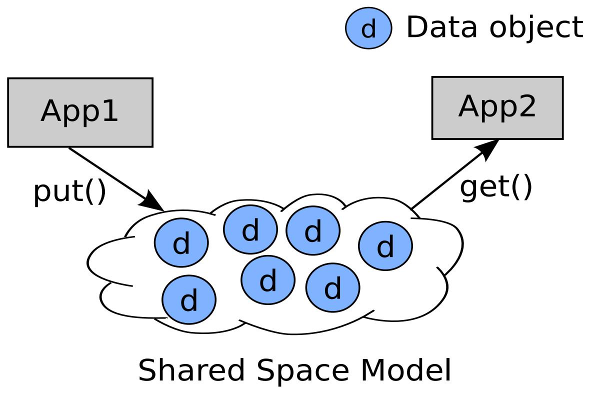


Figure 4.1: Illustration of data sharing through the shared space abstraction.

Shared Space Model is conceptually based on the tuple space model [41] and provides the abstraction of a shared space of data objects which can be associatively accessed by the workflow applications. Data in the space is abstracted as a tuple or object, and each object is associated with a key. Most scientific applications represent data as multidimensional array defined with Cartesian coordinate system. In this case, key associated with a object can be defined as the multidimensional bounding box that describes the array region stored by the object. As illustrated in Figure 4.1, applications can access data objects in the shared space through one-sided *put()/get()* operators.

Array Data Model builds on top of the shared space of objects, and provides the abstraction of a shared space of multidimensional arrays. Applications can directly access the array region of interest through the data insert/retrieve queries with spatial constraints, as illustrated in Figure 4.2. In this case, application only need to express what sub-array it is writing or reading, and does not need to handle details such as how many objects are used to store the sub-array data and where the objects are located. This higher level data abstraction provides a more natural way for workflow applications to share and exchange array-based scientific data.

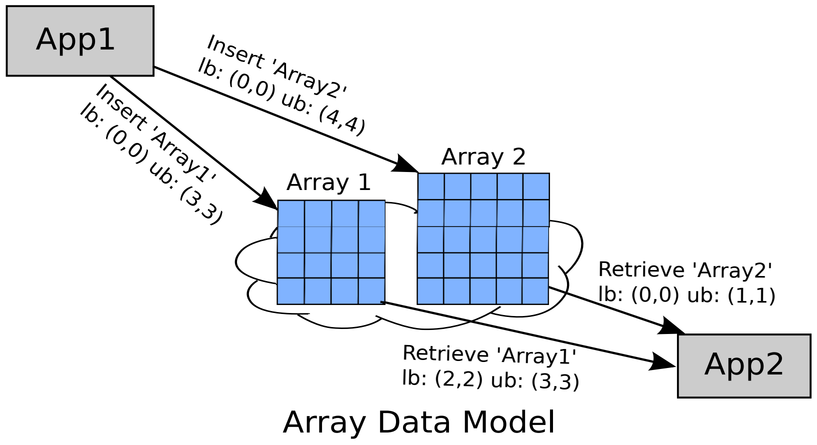

App1
Insert 'Array2' lb: (0,0) ub: (4,4)
Insert 'Array1' lb: (0,0) ub: (3,3)
Array 1
Array 2
Retrieve 'Array2' lb: (0,0) ub: (1,1)
Retrieve 'Array1' lb: (2,2) ub: (3,3)
App2
Array Data Model

Figure 4.2: Illustration of sharing two 2D arrays through the array data model. *lb, ub* denote the bounding box of the inserted/retrieved subarray.

4.3 System architecture

DIMES builds on DataSpaces and inherits its client-server architecture. DIMES clients build a distributed in-memory object store on the application compute nodes to cache data that needs to be shared and exchanged. Applications data is typically represented as multi-dimensional arrays, and each DIMES object stores the data of a subarray. DIMES servers run on dedicated compute nodes, and implement a distributed indexing service to support fast lookup of in-memory data objects. The distributed indexing service enables coupled applications to perform array-based spatial queries to retrieve data from the object store.

Figure 4.3 presents a schematic view of DIMES system architecture. DIMES client and server components build on a common communication layer - DART [36]. DART defines portable communication primitives for asynchronous messaging and data transfers, and implements these primitives using native network programming interfaces such as Cray uGNI, Infiniband verbs, IBM DCMF and PAMI. DART provides low-latency and high-throughput data transfer performance on high-end computing systems. As shown in the figure, DIMES inherits DataSpaces's client and server implementation, and retains all the

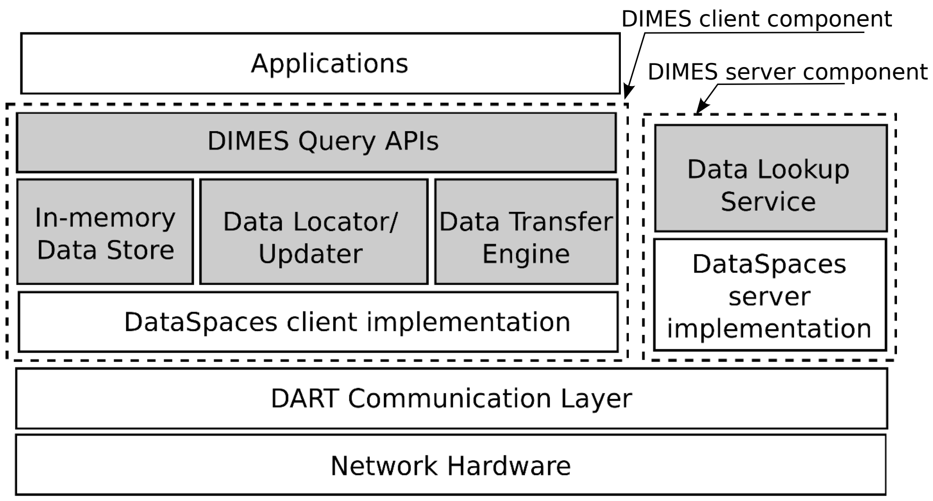


Figure 4.3: DIMES system architecture. Shadowed boxes denote the core functional modules implemented for DIMES.

features supported by DataSpaces.

DIMES servers implement the following core functional module:

- **Data lookup service** enables fast lookup of data objects for data retrieve query. This service constructs a specialized index based on in-memory object's spatial attribute, e.g., a Cartesian bounding box describing the subarray stored by the object. The data lookup service builds its index on the distributed DIMES servers. Global dimension of each newly inserted array is partitioned into non-overlapping regions of equal size. Each DIMES server is assigned one array region, and only maintains indexes for data objects (subarrays) that overlap with the assigned array region.

DIMES clients implement the following core functional modules:

- **Query APIs** defines the set of programming interface that used by applications to share and exchange array-based scientific data. Detailed description of programming APIs and examples is presented in Chapter 4.5.

- **In-memory data store** creates a temporary in-memory object store on application compute nodes. This module manages the physical byte-addressable shared memory segments allocated for the object storage, performs memory allocation for newly inserted object and deallocates evicted object. In addition, this module manages the

RDMA registration for the shared memory segments, which enables RDMA-based one-sided access to the in-memory data objects.

- **Data locator/updater** interacts with the server-side data lookup service to locate objects that need to be fetched for data retrieve query. It also updates the distributed indexing when new data objects are inserted by the applications or existing data objects are deleted from the in-memory object store.

- **Data transfer engine** manages the data transport for data retrieve query. Completion of a retrieve query requires fetching data of one or multiple objects and assembling the fetched data into a contiguous memory buffer provided by the application. Data transfer engine dynamically selects the appropriate data transport mechanism depending on the locations of the data object.

4.4 Implementation

The design objective of DIMES is to build distributed data staging that is co-located on application's compute nodes, and provide the array-based spatial query interface for workflow applications to share and exchange data. This section presents the technical implementation details that enable this design objective.

4.4.1 Node-local in-memory object storage

In the current implementation, each application process runs as a DIMES client, and implements a node-local in-memory object store to cache data inserted by the application. For compute nodes that execute multiple application processes, multiple instances of DIMES clients and object stores co-exist on the same compute node. Figure 4.4 presents a high-level view of the implementation, which is composed of *Memory Management* and *Object Management* subsystems.

DIMES client-side *Memory Management* manages a number of byte-addressable shared memory segments as the storage space for data objects. DIMES uses POSIX shared memory programming interface to create, open, map and remove shared memory segments, which is

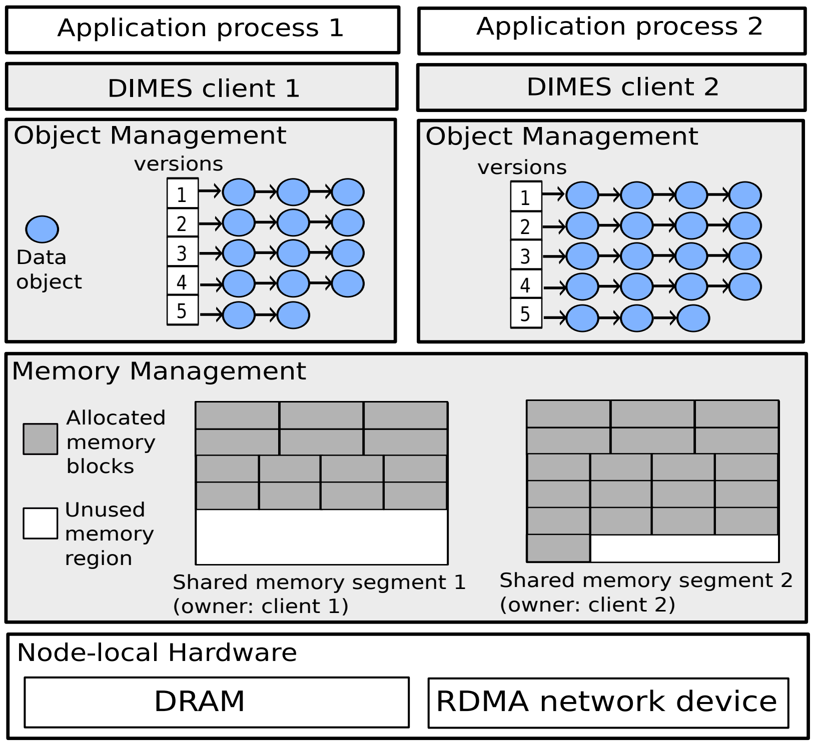


Figure 4.4: DIMES node-local in-memory object storage implementation.

portable and available on most Linux distributions. Data object stored by DIMES is *write-once read-only*, and each DIMES client inserts application data to its local object store. As a result, read access to a shared memory segment is granted to all DIMES clients running on local compute node, but write access is only granted to DIMES client who creates the shared memory segment. For example, as illustrated in Figure 4.4, the compute node has two shared memory segments. DIMES client 1 and client 2 create shared memory segment 1 and 2 respectively, and have read access to both memory segments. However, only DIMES client 1 has write access to shared memory segment 1, and client 2 has write access to shared memory segment 2.

Memory Management implements a customized memory allocator, which allocates memory blocks from the shared memory segment to store newly inserted data objects, and deallocates memory blocks for evicted/deleted data objects. In addition, *Memory Management* manages memory registration/deregistration to support RDMA-based data transport for inter-node data movement. Most high-performance network requires explicit registration of a memory buffer (using the RDMA network programming interface), in order to enable one-sided remote access to the buffer. DIMES client can perform registration using two different approaches: (1) register each allocated memory block on-demand; (2) register the entire shared memory segment in advance. The latter approach is preferred, because it reduces the overhead by minimizing the number of required RDMA buffer register/deregister operations.

Object Management caches multiple versions of a data object in the node-local memory space, and the maximum number of distinct versions can be defined by the programmer. For example, in Figure 4.4, the number of distinct versions is set as 5, and *Object Management* creates 5 slots where each slot uses a linked list to store the data objects. A new data object is inserted to a slot according to the object's version. In the current implementation, we apply a $object\_version \mod num\_distinct\_version$ operation to decide in which slot to insert new data object. *Object Management* also implements a versioning based garbage collection mechanism to delete data objects with older versions. *Object Management* only caches data objects of the most recent N versions, where N is the maximum number of distinct versions. For example, when versions $2N + 1$ and $2N + 2$ of a data object is inserted, versions $N+1$ and $N+2$ are deleted from the in-memory object store. In practice, most coupled simulation workflows consists of applications that run iteratively and need to share data generated in recent iterations. This mechanism works effectively for workflows presenting such data interaction pattern.

4.4.2 Data lookup service

DIMES servers build a lookup service to enable fast lookup of in-memory data objects, specifically for spatial queries that request a spatial region of the cached array data.

Central to the implementation is distributed indexing of in-memory data objects. The key space of the index derives from the global dimension of the multidimensional array variables inserted by application. DIMES uses the Hilbert space filing curve (SFC) to linearize the N-dimensional array domain into a 1-dimensional key space. Using this linearization, a data cell of the array can be uniquely identified using a point in the key space, while a spatial region of the array can be described by a set of intervals in the key space. As a result, a N-d spatial query over the multidimensional array data is translated into a 1-d range query.

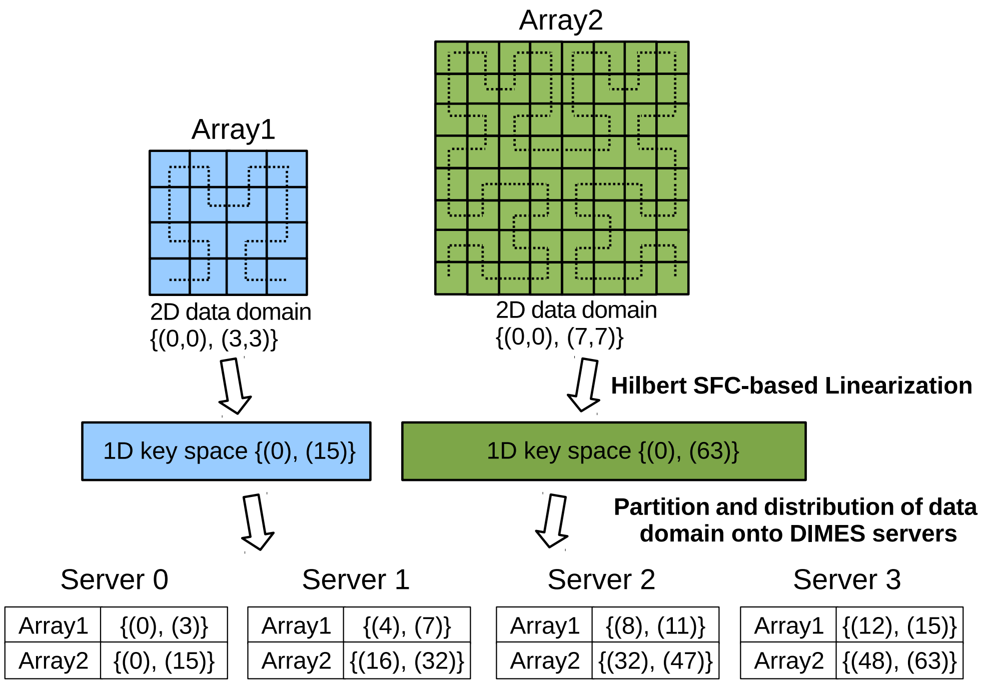


Figure 4.5: Build distributed indexing: (1) construction of index key space by linearization of application data domain using Hilbert SFC (shown as dotted lines); (2) partitioning and distribution of the key space to DIMES servers.

Figure 4.5 illustrates the construction of the 1-d key space. Array1 is a 2D 4×4 array, and its linearized key space can be described by spatial bounding box $\{(0),(15)\}$. Array2 is a 2D 8×8 array, and its linearized key space is $\{(0),(63)\}$.

Linearized key space is partitioned and distributed to DIMES servers, which distributes the objects indexing and query processing workload. Each DIMES server is assigned a set of intervals from the 1-d key space, and only maintains indexing for data objects whose linearized spatial bounding box overlaps with the key space intervals assigned to the server. Similarly, the server only handles data queries whose linearized spatial constraints overlaps with the assigned key space intervals. The current prototype implementation evenly partitions the 1-d key space into N intervals, where N is the number of servers, and each sever is assigned a contiguous key space interval. This partitioning and distribution approach works effectively when the data query load is evenly distributed over the entire data domain and each query fetches a subarray of modest size. More complex partitioning and distribution approaches can be implemented, e.g., distribute the key space based on application's query pattern, but this is beyond the scope of this thesis and can be investigated in future research work.

4.4.3 Data query processing

Application workflows can use DIMES insert/retrieve query interface to exchange array data, and express the array region of interest by specifying the spatial constraint, i.e., a N-dimensional bounding box. This section describes the processing of data insert/retrieve queries.

Data insert query

A data insert query works as follows: (1) Client creates a new data object by allocating a memory block from the shared memory segment, and copies application data into the allocated memory block. (2) Client inserts the new data object into the node-local in-memory object store. (3) Client selects the index servers based on the spatial constraint specified in the query, and sends data update requests to the selected servers. Data update request contains important metadata, e.g., data location, shared memory and RDMA buffer information for the data object. (4) Index servers update the local index for newly inserted data object, and stores the metadata of the object.

Data retrieve query

A data retrieve query involves the following steps: (1) Client selects the index servers based on the spatial constraint specified in the query, and sends data lookup requests to the selected servers. (2) Index servers performs local lookup, and responds to the querying client with metadata for the data objects that need to be fetched. (3) After gathering all the required metadata for the query, client fetches data from local or remote nodes and returns the result data buffer to the user application.

4.5 Programming interface and examples

This sections describes the DIMES C programming interface (see Listing 4.1) that enables array-based data insert and retrieve. It also presents programming examples to illustrate the use of DIMES APIs to implement inter-application data sharing and exchange.

Listing 4.1: DIMES C programming interface.

```
// Define the global dimension for array variable.
void dimes_define_gdim (const char *var_name,
        int ndim, uint64_t *gdim);

// Data insert query.
int dimes_put(const char *var_name, unsigned int ver, int size,
                int ndim, uint64_t *lb, uint64_t *ub, void *data);

// Data retrieve query.
int dimes_get(const char *var_name, unsigned int ver, int size,
                int ndim, uint64_t *lb, uint64_t *ub, void *data);
```

- *dimes_define_gdim()* defines the global dimensions of the array-based data variable to be inserted or retrieved. Programmer-provided global dimension information is used to achieve a balanced distribution of array indexing across DIMES servers.
  - Input arguments: (1) *var_name*: name of the variable. (2) *ndim*: number of array dimension. (3) *gdim*: size in each dimension.
- *dimes_put()* inserts array data into the distributed in-memory object store.
  - Input arguments: (1) *var_name*: name of the variable. (2) *ver*: version of the data. (3) *size*: data size (in bytes) of the array element. (4) *ndim*: number of array dimension. (5) *lb*, *ub*: specify spatial constraint for the query, using a Cartesian bounding box to describe the spatial region of the inserted or retrieved array data. (6) *data*: pointer to user data buffer.
- *dimes_get()* retrieves array data from the distributed in-memory object store.
  - Input arguments: see *dimes_put()*.

Listing 4.2 and 4.3 presents the code snippets using DIMES APIs to share data between applications. The example has two applications, a data producer (see Listing 4.2) and a data consumer (see Listing 4.3). The applications use the locking service implemented by DataSpaces to ensure exclusive access to the shared resource - data variable "particles". The data producer application inserts a 2D 20×20 double array, with spatial bounding box $\{(0,0),(19,19)\}$. The data consumer application retrieves a 10×10 sub-region of the inserted array data, using spatial bounding box $\{(0,0),(9,9)\}$.

Listing 4.2: Data insertion example using DIMES.

```
// Acquires the write lock for variable "particles".
dspaces_lock_on_write("particles");

// Define the bounding box.
gdim[0] = 20; gdim[1] = 20;
lb[0] = 0; lb[1] = 0;
ub[0] = 19; ub[1] = 19;
dimes_define_gdim("particles", 2, gdim);
dimes_put("particles", version, sizeof(double), 2, lb, ub, data);

// Releases the write lock for variable "particles".
dspaces_unlock_on_write("particles");
```

Listing 4.3: Data retrieval example using DIMES.

```
// Acquires the read lock for variable "particles".
dspaces_lock_on_read("particles");

// Define the bounding box.
gdim[0] = 20; gdim[1] = 20;
lb[0] = 0; lb[1] = 0;
ub[0] = 9; ub[1] = 9;
dimes_define_gdim("particles", 2, gdim);
dimes_get("particles", version, sizeof(double), 2, lb, ub, data);

// Releases the read lock for variable "particles".
dspaces_unlock_on_read("particles");
```

4.6 Experimental evaluation

The prototype implementation of DIMES was evaluated on the Cray XK7 Titan system at Oak Ridge National Laboratory. Titan has 18,688 compute nodes, and each compute node has a 16-core AMD Opteron processor, 32GB memory, and a Gemini router that interconnects the nodes via a fast network with a 3D torus topology.

This sections presents two distinct sets of experiments. The first set of experiments evaluates the performance and scalability of DIMES insert/retrieve queries with increasing numbers of application processes and data sizes. The second set of experiments presents the integration of DIMES with a coupled fusion simulation workflow, and compares the data exchange performance with file-based and server-based approaches.

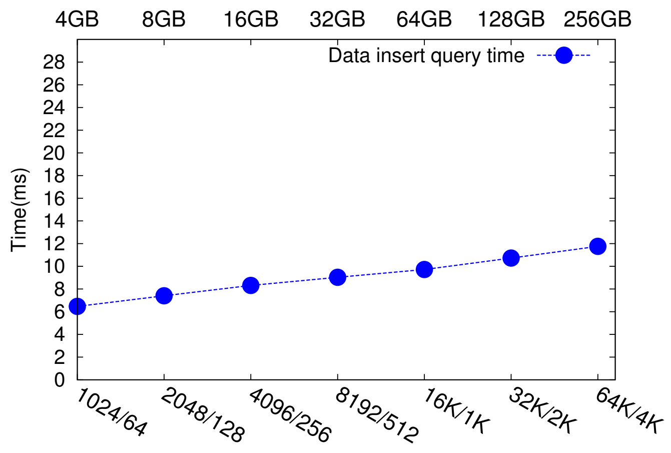
4.6.1 Evaluation of scalability

Coupled simulation workflows simulating complex phenomena such as climate modeling and plasma fusion science typically run on a large number of processor cores and significant amounts of data is transferred between the component applications. The scalability experiments presented in this section evaluate the ability of the DIMES framework to support parallel data redistribution for a different numbers of application processes and data sizes.

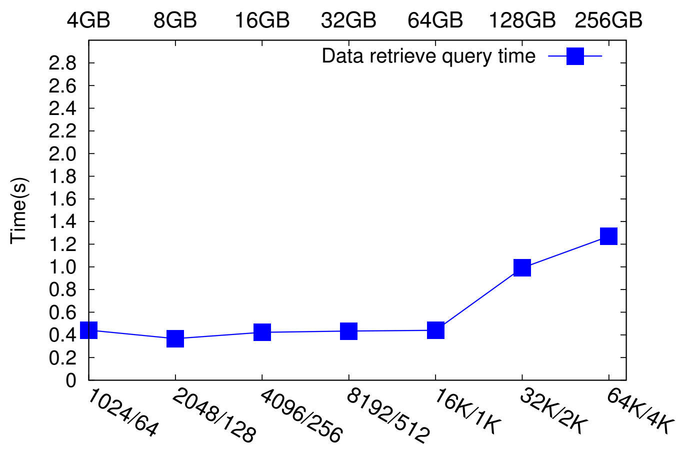
This experiment uses a testing workflow composed of two applications, a *Writer* and a *Reader* application, which captures data sharing and exchange behaviors while removing the complexity of the computational aspects of the full simulation codes. The two applications perform parallel computations over a common 3-dimensional computational domain, and each application is assigned a distinct set of processor cores. The coupled variable is a 3-dimensional global array, which is decomposed using a standard blocked distribution. During the workflow execution, DIMES is used to support the parallel data redistribution which transfers and redistributes the global array data from *Writer* application processes to the *Reader* processes.

The ratio of *Writer* to *Reader* application processes is fixed as 16:1. The size of the *Writer* application is varied from 1K to 64K processes. Meanwhile, the size of *Reader* application is varied from 64 to 4K processes. The number of DIMES servers is varied from 4 to 256, which uses less than 0.4% of the total compute nodes allocated for workflow execution.

The experiment evaluates both weak scaling and strong scaling performance. In the weak scaling experiment, each *Writer* process inserts a fixed 4MB of data per iteration, and the total data size is varied from 4GB to 256GB. In the strong scaling experiment, the total data size being transferred is fixed as 4GB. In each experiment, the applications ran for 200 iterations to simulate 200 parallel data redistributions between the *Writer* and *Reader* applications, and the average data insert and retrieve query time is presented.



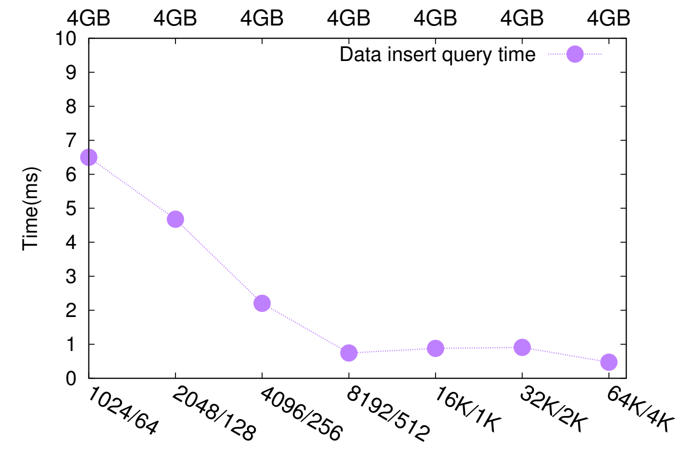
(a) Data insert query performance



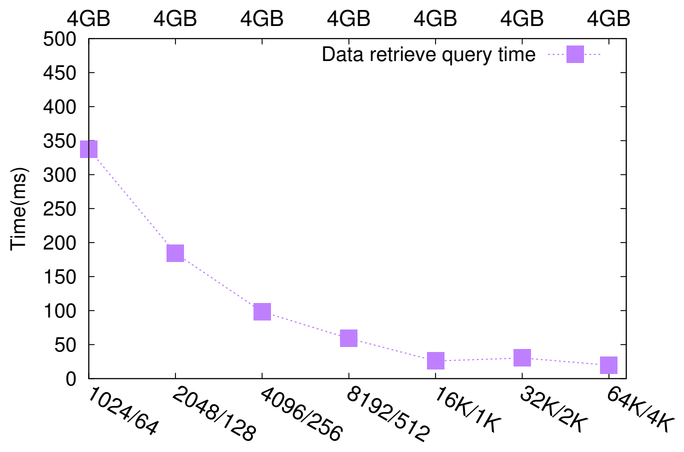
(b) Data retrieve query performance

Figure 4.6: Weak scaling results on Titan. The bottom X axis is the size of *Writer* application / the size of *Reader* application. The top X represents the size of data that is transferred and redistributed in each iteration.

Figure 4.6 presents the performance of data insert and retrieve queries for a weak scaling



(a) Data insert query performance



(b) Data retrieve query performance

Figure 4.7: Strong scaling results on Titan. The bottom X axis is the size of *Writer* application / the size of *Reader* application. The top X represents the size of data that is transferred and redistributed in each iteration.

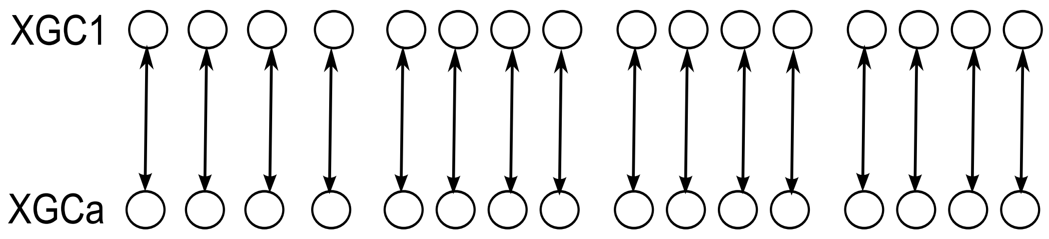
experiment. The results show good overall scalability with increasing number of processes and data sizes. Because a DIMES client writes data into node-local memory region, the data insert query does not need to perform any off-node network data movement and is very fast. The average data insert query time is between 6 and 12 ms, which has minimal overhead on the *Writer* application. Data retrieve query time sustains at about 0.4 seconds, as the size of workflow is increased from 1K/64 to 16K/1K. Data retrieve query time increases to 0.99 seconds at 32K/2K, and to 1.27 seconds at 64K/4K. The performance degradation of data retrieve query is mainly due to the increased contention at the shared network links, which is caused by the increasing number of concurrent data transfers at larger application sizes. However, this small increase in transfer time is acceptable when considering the scale of the application and the total data size.

Figure 4.7 presents the performance of data insert and retrieve queries for a strong scaling experiment. The results show good strong scaling performance. As the size of workflow increases from 1K/64 to 8K/512, the data query performance scales linearly. Average data insert query time decreased by about 8 times from 6.49 ms to 0.74 ms, and the average data retrieve query time decreased by about 6 times from 337.58 ms to 59.12 ms. Both curves become flat when the size of workflow is further increased. Though the size of data inserted or retrieved by each process keeps decreasing in the strong scaling experiment, there exist costs that are independent of the data size and can not be further reduced, such as the cost of locating data in a retrieve query, and updating data in an insert query. As a result, both curves become flat when the size of workflow is between 16K/1K and 64K/4K.

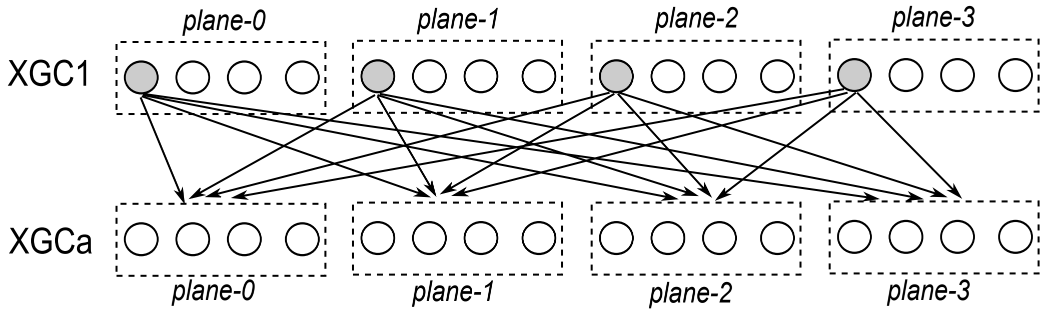
4.6.2 Evaluation with coupled fusion simulation workflow

Overview

This section describes using DIMES to support the in-memory data sharing for a coupled plasma fusion workflow. The workflow provides insight into plasma edge physics in magnetic fusion devices, and consists of two separate axisymmetric kinetic transport codes XGC1 and XGCa [48, 30]. The workflow executes for multiple coupling iteration. In each coupling iteration, the workflow first executes XGC1 for n time steps to compute turbulence data and particle state, and then executes XGCa for m time steps to evolve the state of plasma. The sharing of turbulence data is one-way from XGC1 to XGCa. XGC1 writes turbulence data at each time step of its execution, which is read by XGCa in the subsequent execution step. The sharing of particle data is two-way. Both XGC1 and XGCa write particle data at the end of their executions, and needs to read particle data generated by the other application at the beginning of their executions.



(a) Sharing particle data between XGC1 and XGCa processes.



(b) Sharing turbulence data between XGC1 and XGCa processes.

Figure 4.8: Illustration of communication pattern for sharing particle and turbulence data in XGC1-XGCa coupled simulation workflow.

Figure 4.8(a) illustrates the communication pattern for sharing particle data. Communication pattern for sharing particle data depends on the decomposition and distribution of plasma particles across the application processes. Typically, XGC1 and XGCa have the same decomposition and distribution for particles. As a result, the sharing of particle data requires simple *one-to-one* communication between XGC1 and XGCa processes.

Figure 4.8(b) illustrates the communication pattern for sharing turbulence data, which depends on the workflow's poloidal domain decomposition. In this simple example, XGC1 and XGCa executes on 16 processes and has 4 poloidal planes namely *plane-0* to *plane-3*, where each poloidal plane has 4 application processes. During the execution of XGC1, each poloidal plane selects one process (marked with gray color in the figure) to write the plane's turbulence data. During the execution of XGCa, each application process of poloidal plane

i needs to read the turbulence data from four poloidal planes including $i-1$, i, $i+1$ and $i+2$. For example, XGCa processes in *plane-1* need turbulence data generated by XGC1's *plane-0*, *plane-1*, *plane-2* and *plane-3*. The sharing of turbulence data exhibits *one-to-many* communication between XGC1 and XGCa processes.

Results

Prototype implementation of the in-memory data sharing utilizes DIMES to enable distributed in-memory data staging and sharing. XGC1 and XGCa executes on the same set of compute nodes, and writes turbulence and particle data into node-local memory buffer that is managed by DIMES.

Table 4.1 summarizes the experimental setup. The number of application processes for XGC1/XGCa is varied from 1K to 16K. The workflow runs for 2 coupling iteration, both XGC1 and XGCa executes for 20 time steps per coupling iteration. In each coupling iteration, the size of particle data read by XGC1 and XGCa ranges from 4GB to about 65GB, and the total size of turbulence data read by XGCa ranges from 12GB to about 202GB. To demonstrate the benefits of the DIMES approach, we compare the performance with both the *file-based* approach, which exchanges data through disk files, and the *server-based* approach, which exchanges data through a set of dedicated staging compute nodes.

	Setup 1	Setup 2	Setup 3
Num. of processor cores	1024	4096	16384
Num. of coupling iteration	2	2	2
XGC1 num. of steps (per iteration)	20	20	20
XGCa num. of steps (per iteration)	20	20	20
Size of particle data written/read by XGC1 (per iteration)	4.05 GB	16.21 GB	64.85 GB
Size of particle data written/read by XGCa (per iteration)	4.05 GB	16.21 GB	64.85 GB
Size of turbulence data write by XGC1 (per iteration)	0.19 GB	0.19 GB	0.19 GB
Size of turbulence data read by XGCa (per iteration)	12.63 GB	50.52 GB	202.09 GB

Table 4.1: Experimental setup for the evaluation of XGC1-XGCa coupled simulation workflow.

Tables 4.2 and Figure 4.9 presents the performance for exchanging particle data between

XGC1 and XGCa applications. The results show significant performance improvement for DIMES when compared with the two other approaches. As shown by Tables 4.2, DIMES decreases the total time for writing particle data by 99% on average when compared with *file-based* approach, by 93% on average when compared with *server-based* approach. As shown by Figure 4.9, DIMES decreases total time for reading particle data time by 98% on average when compared with *file-based* approach, by 92% on average when compared with *server-based* approach. This significant performance advantage is due to DIMES's in-memory local data caching, which does not require moving data off-node. In the *file-based* approach, particle data needs to be moved and written to storage system, while in the *server-based* approach, particle data needs to be transferred to the staging servers.

	Setup 1	Setup 2	Setup 3
File-based (seconds)	35.792	90.062	425.059
Server-based (seconds)	1.865	2.283	3.781
DIMES (seconds)	0.097	0.131	0.316

Table 4.2: Total time of writing particle data.

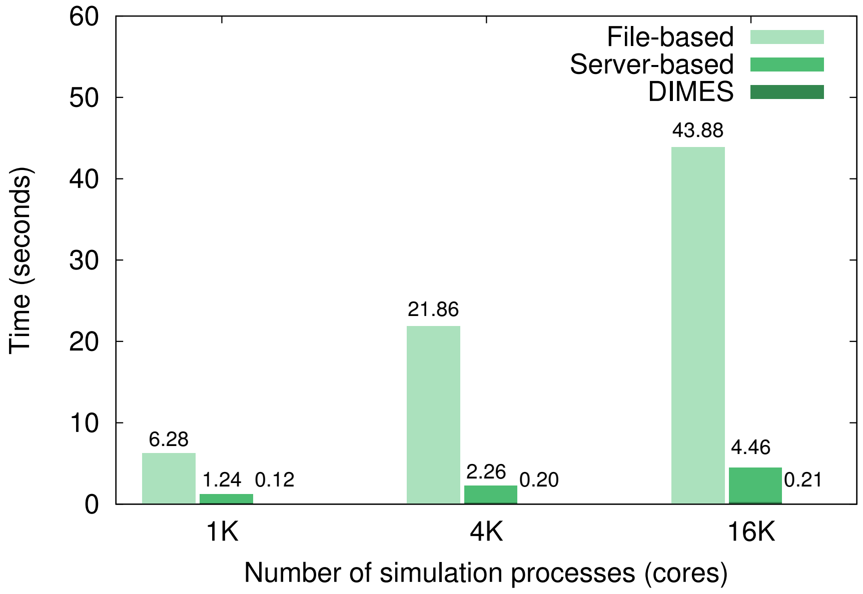


Figure 4.9: Total time of reading particle data.

Table 4.3 and Figure 4.10 presents the performance of exchanging turbulence data between XGC1 and XGCa. As shown by Table 4.3, DIMES decreases the total time for writing turbulence data by 99% on average when compared with *file-based* approach, and by 31%

on average when compared with *server-based* approach. One observation is that when compared with the *server-based* approach, the performance improvement of writing turbulence data is not as significant as that of writing particle data. The reason is that the size of turbulence data written by XGC1 application is very small (as can be seen in Table 4.1). As a result, the time for transferring turbulence data to staging servers is much smaller. Figure 4.10 presents the performance for reading turbulence data. DIMES decreases the total time for reading turbulence data by 99% on average when compared with *file-base* approach, and by 96% on average when compared with *server-based* approach.

	Setup 1	Setup 2	Setup 3
File-based (seconds)	36.228	31.236	16.430
Server-based (seconds)	0.024	0.039	0.029
DIMES (seconds)	0.016	0.029	0.019

Table 4.3: Total time of writing turbulence data.

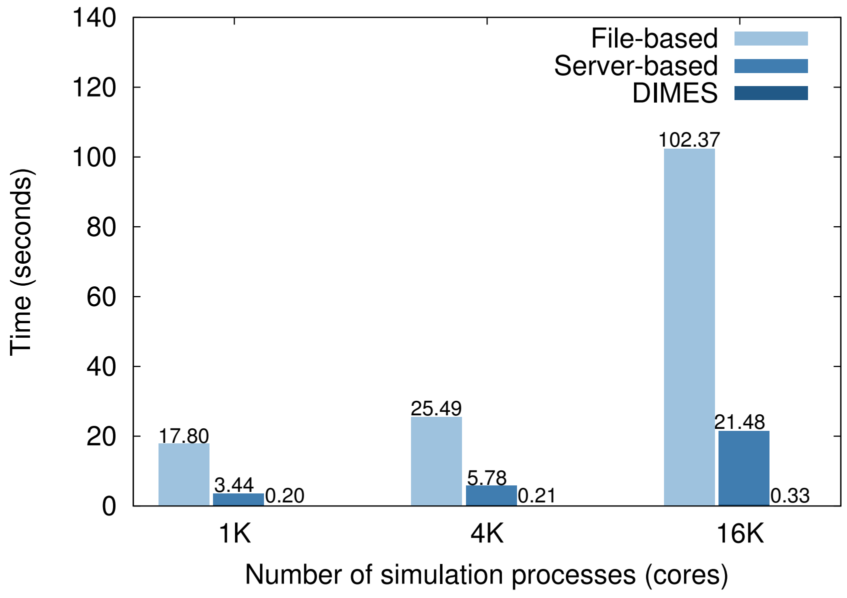


Figure 4.10: Total time of reading turbulence data.

4.7 Related work

4.7.1 Programming abstractions for expressing parallel data exchange

Coupled simulation workflow consists of multiple applications that need to interact and exchange data at runtime, and requires the programming support for expressing data exchange between coupled applications. Generic low-level parallel programming abstractions, such as message-passing interface [42], can be used to implement the data exchange. However, it is nontrivial to use such low-level communication interface, and requires programmers investing significant effort into every implementation detail required by the parallel data exchange, such as memory management, data indexing and lookup, scheduling data transfers. This section presents research work that provides high-level programming and data abstraction, which is more expressive and hides the low-level implementation details.

Shared tuple space abstraction provides the abstraction of a shared repository of tuple objects, which can be associatively accessed. Linda [41] programming language initially introduces tuple space as the coordination model for data sharing and communication between parallel processes, and implements a set of primitives, e.g. *in()*, *out()*, *rd()*, *eval()*, to access the tuple space. DataSpaces [35] and Seine [71] build on the concept of tuple space abstraction and implement semantically specialized data space to support coordination and data exchange between applications that are part of a coupled simulation workflow. In particular, DataSpaces provides the global array based data abstraction to facilitate exchanging distributed scientific array between coupled applications. DIMES presented in this thesis implements the shared data space abstraction that is similar to DataSpaces.

Common component architecture (CCA) [14] is an effort to support interoperability between coupled scientific applications, which promotes the concept of component object model that is similar in industrial standards such as CORBA [2] and COM [1]. Scientific application is abstracted as a component and implements a standard set of interfaces. Applications interact by invoking the remote component interfaces. Specifically, CCA defines the *collective ports* interface to support data exchange between parallel components that run on multiple processes. Several CCA-compliant runtime frameworks have been developed, including InterComm [50], Parallel Application Work Space (PAWS) [40], MetaChaos [38]. CCA approach only supports data exchange between concurrently running applications, because it requires the presence of remote component application in order to establish connection and invoke the component interfaces. Sharing data between sequentially running applications can not be supported by CCA approach. In contrast, DIMES shared data space abstraction decouples the data sharing from application execution, and supports data exchange for both concurrently and sequentially running applications.

Partitioned global address space (PGAS) has emerged as a promising parallel programming model, which provides a global memory address space that is logically partitioned in the context of distributed memory machines. PGAS languages and libraries, such as UPC [22], Chapel [25], X10 [26], GlobalArray [56], supports the global-view multidimensional array, which provides a convenient high-level data abstraction to share data between application processes or threads. While existing PGAS implementations enable sharing data between parallel processes of single application, they do not support data coupling across heterogeneous programs. For example, transferring and redistributing global-view array between different application programs is not supported by existing PGAS languages and runtime systems. DIMES provides a shared space abstraction that is globally accessible by applications that are part of a workflow, and supports inter-application data sharing and exchange.

Flexpath [29] uses a type-based publish/subscribe approach to implement communications between simulations and concurrent running data analytics. The publish/subscribe pattern decouples the analytics services from simulation codes, and supports dynamic arrivals/departures of analytics services. Flexpath essentially implements the data streaming model between data producer and consumer applications. However, Flexpath is limited to workflow scenario where data producer and consumer applications need to run concurrently, and does not support workflows that couple sequentially executed producer and consumer applications.

4.7.2 Parallel data exchange for coupled applications

One intuitive and simple approach to exchange data between coupled applications running on HPC cluster is sharing data through the distributed file systems, such as Networked File System (NFS), Parallel Virtual File System (PVFS) [23], General Parallel File System (GPFS) [62], Lustre [5]. This approach enables applications to use common data access interfaces (e.g. POSIX I/O) and a wide variety of scientific data formats (e.g. HDF5, NetCDF), which brings productivity, portability and flexibility to the software development. However, the increasing performance gap between computation and disk I/O introduces significant data sharing overhead and limits the data exchange performance. This section presents systems that support memory-based data sharing and exchange, which can be categorized into two classes.

Direct data movement between coupled applications: Model Coupling Toolkit (MCT) [49] is a Fortran-based software package to couple MPI-based component models into a single coupled model. MCT requires running the workflow as a single MPI executable, and partitions the MPI_COMM_WORLD global communicator into a set of sub-communicators for each of the component model. MCT implements the direct data transfers between component models using the inter-communicator communications supported by MPI. Flexpath [29] implements a serverless publish/subscribe system to support direct connection and communication between coupled applications. Flexpath implements direct data transfers by leveraging multiple communication mechanisms, including shared memory, Remote Direct Memory Access (RDMA), TCP/IP, which provides high performance and portability. Direct data movement approach avoids extra data transfers when compared with approaches that require moving data to external servers. However, existing implementations of direct data movement approach only support coupling concurrently running applications, and does not support sharing data between sequentially running applications.

Inter-application data movement through dedicated staging servers: DataSpaces [35] builds a distributed in-memory storage space on a set of dedicated servers. DataSpaces stores and indexes data inserted by the application, and provides a query API to retrieve the data cached in server memory. In addition, it leverages RDMA one-sided communication to implement high-performance data transfers between application and servers. H5FDdsm [19] framework builds around the idea of virtual file in distributed shared memory, and enables coupled applications use HDF5 I/O API to exchange data. Virtual file is stored in a distributed shared memory buffer allocated on a set of severs. Using the HDF5 virtual file driver extension, H5FDdsm transparently reroutes all I/O requests to the in-memory file. Staging-based data movement uses dedicated servers to store data. The availability of data is not dependent on the presence of data producer application as in the direct data movement approach, which enables data sharing that is decoupled in time. However, staging-based approach requires transferring data twice, first from data producer application to server, then from server to data consumer application, which may increase the latency.

4.8 Summary

This chapter presented the design, implementation and evaluation of DIMES - a data management framework for supporting memory-based data sharing and redistribution in coupled scientific simulation workflow. DIMES co-locates distributed data staging with application execution, and caches application data in node-local object store. This approach effectively reduces the volume of network data movement when compared with data coupling approaches that cache data on remote storage servers. In addition, DIMES indexes staged data objects according to their spatial attributes, and the array-based query interface enable applications to retrieve data of interest. DIMES utilizes high-performance hardware-enabled RDMA operations for inter-node communication, and on-node shared memory for intra-node communication, which provides efficient and scalable data movement. This chapter also presented evaluation results on large-scale HPC cluster, and demonstrated the scalability and performance of DIMES data insert/retrieve queries.

Chapter 5

CoDS - a framework for workflow task execution and placement

5.1 Introduction

DIMES data management framework (presented in Chapter 4) addresses the problem of supporting distributed data staging, memory-based data sharing and exchange for coupled simulation workflow. This chapter presents CoDS framework to provide the programming and runtime support for *workflow composition*, *task execution* and *task placement*.

CoDS implements a task execution engine to support scheduling and executing parallel task programs. It manages a set of compute nodes allocated for workflow execution, and integrates DIMES to support in-memory data management. CoDS implements locality-aware task placement, which attempts to place task execution on compute nodes that contain the largest portion of its input data. To achieve this objective, CoDS allows programmers to express locality preference for a task by providing *data hint*, e.g., name and spatial region of the array data that to be accessed by the task. In addition, CoDS allows users to programmatically customize and control the task placement, according to the specific need of the targeted workflow. For example, programmers can functionally partition the computation resource, and explicitly express task placement affinity by providing *location hint*, e.g., the preferred functional partition for the task execution.

CoDS workflow representation extends the DAG task graph abstraction used by traditional scientific workflow management systems with data flow annotation, iterative execution, and data-driven task execution, in order to represent the execution model of coupled simulation workflow. CoDS's programming environment is exposed as a library to existing language (current implementation supports C), and implements task execution APIs to compose a workflow. Workflow written for CoDS consists of a driver program and a number of task programs. Each task program represents a component application that is part of the workflow, and is typically based on existing MPI parallel simulation or data analysis code. The driver program is written to compose the actual workflow by combining sequential, concurrent and iterative execution of task programs.

It's worth noting that CoDS is not a system-wide task execution framework deployed on the HPC cluster. Typical HPC cluster uses a system-wide job scheduler, such as SLURM [45], to support resource allocation, scheduling, and execution of jobs submitted by multiple users. CoDS is a light-weight task execution framework, and targets executing coupled simulation workflows using compute nodes in a single job allocation.

Technical contributions of CoDS is summarized as below:

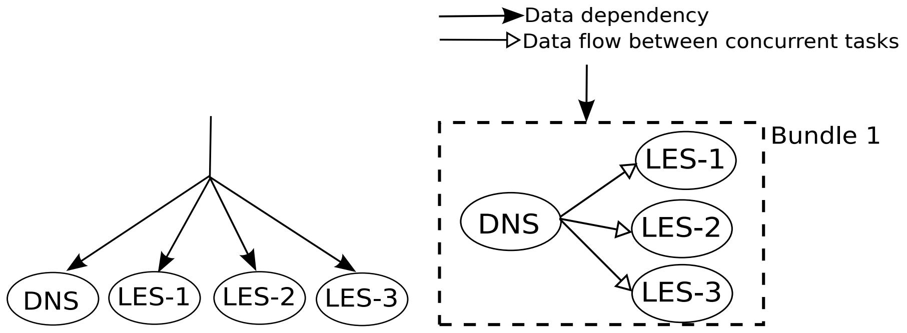
- CoDS presents a highly programmable and customizable task execution framework. CoDS provides the programmability to partition the allocated compute nodes/cores, and create different functional partitions according to the need of the workflow. For example, in online data analytics workflow that requires in-transit placement of analysis operation, users can programmatically divide the processor cores into two partition. One partition of processor cores is used for simulation execution, while the other partition is used for "in-transit" data analysis and processing.

- CoDS allows users to provide hint information for task placement. Users provide task placement hint according to their knowledge and experience. Hint-based task placement mechanism aims at making a better placement decision by combining the programmer-provided and runtime information. Current prototype implementation supports two types of placement hint, including *data hint* that is used by the runtime to perform locality-aware placement, and *location hint* that explicitly specifies the preferred functional partition.

The remainder of this chapter is organized as follows. Section 5.2 presents the workflow representation. Section 5.3 presents CoDS system architecture. Section 5.4 presents the design and implementation. Section 5.5 presents the programming interface and examples. Section 5.6 presents the evaluation using two real world coupled simulation workflows. Section 5.7 presents related research work and Section 5.8 summarizes the chapter.

5.2 Workflow representation

CoDS workflow representation builds on the DAG task graph abstraction used by traditional scientific workflow management systems [3, 32]. Each vertex in the DAG represents an application and each edge represents the data dependency, where an application can start after all its parent applications finish executions. In order to represent the coupled simulation workflow, we extend the basic DAG task graph abstraction in the following ways.



(a) DNS-LES task graph (b) DNS-LES task graph with bundle annotation

Figure 5.1: Task graph representations: DNS-LES workflow.

First, we extend the basic DAG task graph representation with the concept of **bundle** to annotate a group of tightly coupled applications that execute simultaneously and have frequent inter-application data communications. The basic DAG representation can not capture the data streams/flows between *concurrently* executing applications that have no explicit data dependency. For example, DNS-LES workflow for combustion science couples DNS solver with multiple instances of LES solvers. DNS and LES executes concurrently in lockstep, and data is transferred from DNS to LES instances at every sub step, i.e., 6 times in a single time step. Figure 5.1(a) presents the DAG representation for a workflow that couples DNS with 3 LES instances. Figure 5.1(b) presents the same workflow using DAG with bundle annotation. Bundle annotation describes which concurrent applications are tightly coupled as well as the dataflow between concurrent applications.

Second, task graph abstraction needs to represent **iterative execution**, which is required by many simulation workflows. For example, XGC1-XGCa workflow for plasma fusion science requires iterative execution of sequentially coupled XGC1 and XGCa simulation code. Figure 5.2 presents the task graph representation for the XGC1-XGCa workflow.

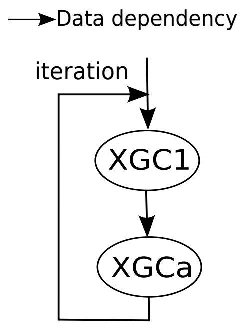


Figure 5.2: Task graph representations: XGC1-XGCa workflow.

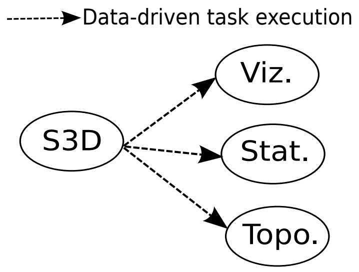


Figure 5.3: Task graph representations: S3D data analytics workflow.

Third, the task graph needs to represent **data-driven task execution**, which starts a task when all its data dependencies are satisfied. This differs from the dependency resolution in traditional scientific workflow management systems [3, 32] where the execution of a task depends on the completion of its parent tasks. The ability of running data-driven task is very useful for online data analytics workflow. For example, S3D data analytics workflow for combustion science couples the simulation code S3D with several analysis operations, e.g., visualization, descriptive statistics, topological analysis. With the data-driven execution capability, runtime system can dynamically start concurrent execution of a analysis task after its required data is produced by S3D. Figure 5.3 presents the task graph representation for S3D data analytics workflow (the dotted arrow shows data-driven coupling between tasks).

5.3 System architecture

CoDS task execution framework consists of four components including *task executor*, *workflow manager*, *task submitter*, and *information space*. Figure 5.4 presents a graphical representation of the system architecture.

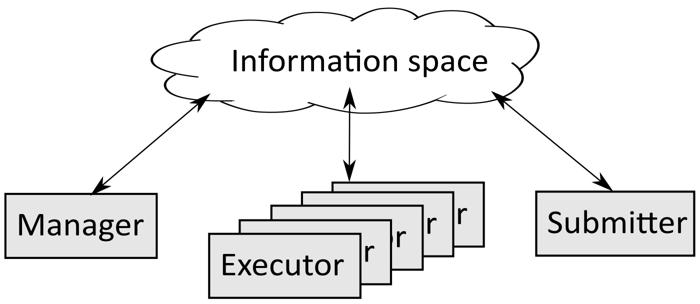


Figure 5.4: Workflow execution framework system architecture.

Task executor is responsible for executing workflow tasks. Each task executor runs on one physical processor core, and a pool of task executors are assigned to the allocated compute nodes. In addition to task execution, the executor also integrates DIMES to cache workflow data in local memory of compute nodes, and support data sharing and exchange between interacting workflow tasks.

Task submitter is responsible for submitting task execution request to the workflow manager. Task submitter executes the driver program at runtime, and drives the workflow progression by launching asynchronous task executions on the executor resource pool.

Workflow manager is responsible for coordinating task executions within a workflow. Workflow manager maintains three tables to respectively store the information about workflow tasks, data variables, and executors. *Task table* contains the information for submitted tasks. *Variable table* contains the information for currently available data variables that appears as a task input or output. *Executor table* contains the information of all task executors in the system. Each task has zero or multiple input dependencies and becomes runnable when all its dependencies are satisfied. Based on the state of *task table* and *variable table*, workflow manager can perform dependency resolution to identify runnable tasks.

Information space runs on a dedicated set of compute nodes, and builds on DataSpaces servers. Information space enables coordination and messaging between other components, and supports indexing and lookup of data cached in DIMES.

5.4 Implementation

5.4.1 Workflow execution

Workflow makes progress by executing tasks. This subsection outlines the key steps of workflow execution.

The first key step is **task submission**. Both task submitter and executor can send task submission requests to workflow manager. Workflow manager adds a new entry to *task table*. Meanwhile, the workflow manager adds input data variables that the task depends on into *variable table*. For each data variable in *variable table*, workflow manager needs to subscribe on the variable's status that is maintained by the information space. Workflow manager will receive notifications from the information space whenever the data variable of interest becomes available, updated or deleted.

The second key step is **dependency resolution**. Workflow manager evaluates if a task is runnable, utilizing information stored in *task table* and *variable table*. When a task becomes runnable, it is ready for scheduling and placement.

The third key step is **task scheduling and placement**. Firstly, workflow manager needs to decide when to execute a runnable task. A runnable task becomes ready for execution when its computation resource requirement is satisfied. Current implementation employs simple first-come-first-serve policy to select the next ready-for-execution task. Secondly, workflow manager needs to decide where to execute the task, by selecting the idle executors for task execution. In current implementation, *executor table* orders the task executors by their compute node id. The node id is system-wide information and typically preserves the network locality of compute nodes. For example, on ORNL Titan system, compute nodes with adjacent id are physically connected with direct network link. By default, the workflow manager allocates contiguous idle executors in the table to execute a task. In addition to the default placement policy, the framework also supports two customized task placement approaches to optimize the workflow execution, which is described in next section.

The last key step is **task execution**. Workflow manager dispatches the task to idle executors selected during task scheduling and placement, then the executors start executing the assigned task. As noted previously, workflow task is typically a MPI parallel program. CoDS runtime needs the ability to dynamically launch and execute a task, i.e., the MPI executable, on selected task executors. However, though MPI standard defines the dynamic process management [43], this feature is not fully supported on many current generation high-end computing systems, such as Cray XE/XK and IBM BlueGene/Q. As a result, current prototype implementation requires to wrap the MPI task program as a library. The execution framework dynamically creates a MPI communicator on the idle executors selected for the task, and starts executing the task by invoking the entry function of the task library.

5.4.2 Hint-based task placement

Task placement determines which physical compute nodes and processor cores to execute a task, and has significant impact on the performance of overall workflow execution. CoDS framework implements hint-based task placement mechanism, which combines programmer-provided placement hint and runtime information (e.g., location of data). Specifically, current prototype implementation supports two types of programmer-provided placement hint, including **data hint** and **location hint**.

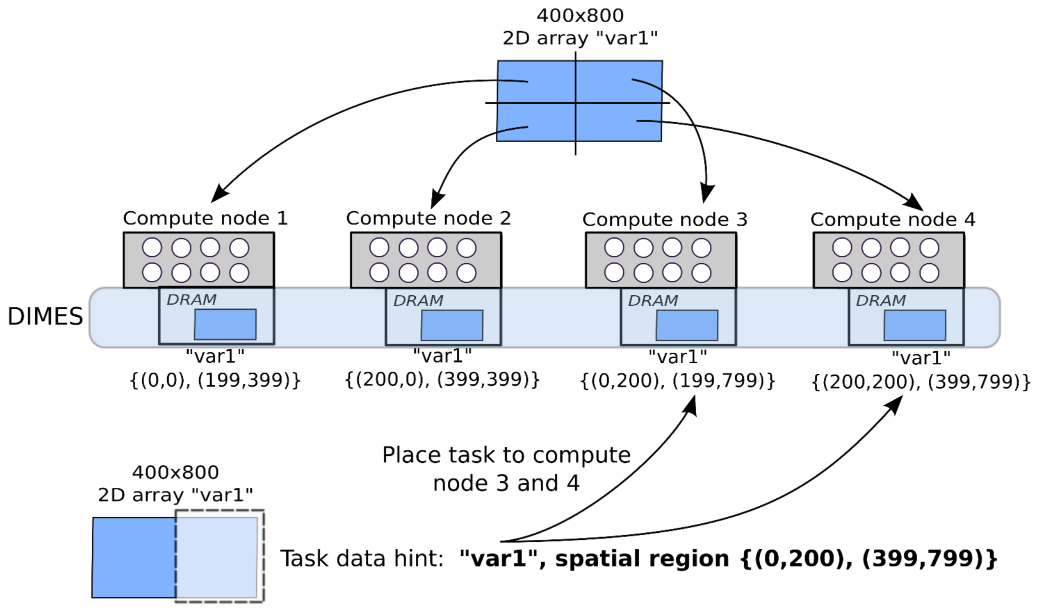


Figure 5.5: Illustration of locality-aware task placement through programmer-provided data hint.

Data hint allows programmers to express locality preference for a task. CoDS framework performs locality-aware task placement according to the data hint, with the objective of moving computation closer to data. Workflow data is staged in memory across the compute nodes, and reading data from local memory requires no network data movement and is much faster than reading data from remote memory. Locality-aware placement aims at executing task on compute nodes where a large portion of input data is available in node-local memory, which can effectively reduce the size of data movement over network. Figure 5.5 presents a simple example of moving task to data by utilizing programmer-provided data hint. Input data variable of the task is a 2D 400×800 array, and the array data is distributed over 4 compute nodes, namely n1, n2, n3, n4. Instead of reading the entire 2D array, the task only needs to read subarray $\{(0, 200), (399, 799)\}$. Programmer can provide the spatial bounding box to describe the array region of interest. At runtime, workflow manager can retrieve the data location information by querying DIMES data lookup service, and places task onto compute nodes n3 and n4, in order to maximize the size of locally available input data.

Location hint allows programmers to explicitly express the functional partition for task execution. This approach first divides the allocated computation resource (or task executors in CoDS) into programmer-defined functional partitions. Each partition represents one possible execution location. Programmers then can provide location hint to specify the preferred partition of executors for a task.

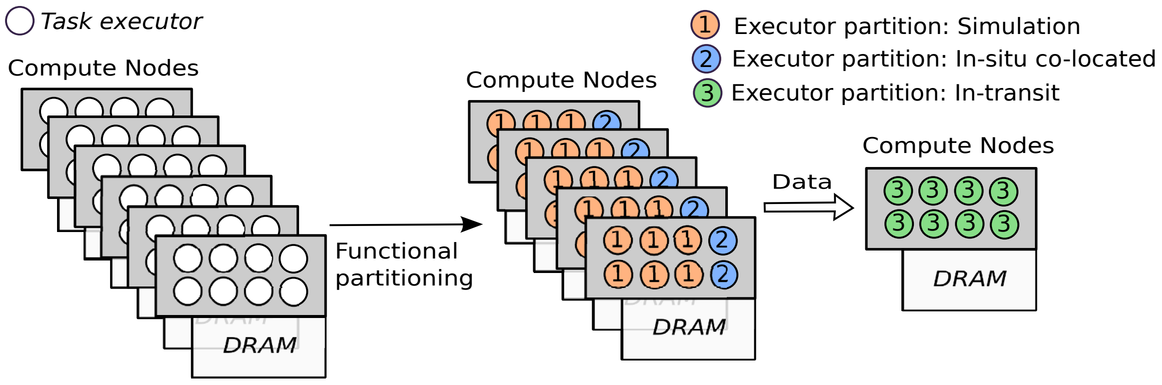

Task executor
Compute Nodes
DRAM
Functional partitioning
Compute Nodes
Executor partition: Simulation
Executor partition: In-situ co-located
Executor partition: In-transit
DRAM
Data
Compute Nodes
DRAM

Figure 5.6: Illustration of functionally partitioning the task executors to create different placement options for analysis operation in online data analytics workflow.

Location hint based task placement targets specifically the online data analytics workflow. In a typical execution of online data analytics workflow, computation resource can be divided into three possible partitions (as shown in Figure 5.6), which creates the following placement options: (1) In-situ inline - data analysis and simulation share the same processor cores. (2) In-situ co-located - processor cores on each compute node are functionally partitioned into simulation cores and analysis cores. Data analysis is executed on the dedicated analysis cores. (3) In-transit - data analysis is executed on a dedicated set of compute nodes. As presented by recent research work [72, 16], data analysis operations may have different placement preference, depending on various factors such as scalability of the analysis algorithm, input data size of the analysis. Support of location hint presents several advantages: First, it enables placement flexibility for data analysis. Programmers can explicitly control the placement of analysis operation according to their expertise knowledge about the workflow. Placement location can be changed simply by changing the location hint, which enables programmers to explore different placement locations and the tradeoffs. Second, it enables customized partitioning of computation resource as required by the specific workflow. For example, some workflows may require the placement options of "in-situ co-located" plus "in-transit", while others may only require the placement option of "in-transit".

5.5 Programming interface and examples

This section presents the programming interface and examples for composing the workflow and providing task placement hint.

5.5.1 Workflow composition

This section presents the task execution APIs, which is used to compose coupled simulation workflows. Programmers uses the API to launch asynchronous task execution on idle executors. Task program can be serial code running on one executor, or parallel code running on multiple executors. Programmer can specify input data dependency for a task, and the CoDS workflow manager performs dependency resolution and starts the task execution when all input data of the task becomes available.

Listing 5.1: Task execution C programming interface.

```
// Initialize the runtime framework.
int cods_init();

// Finalize the runtime framework.
int cods_finalize();

// Add task to a bundle.
int cods_add_task(bundle_descriptor *bundle_desc,
                  task_descriptor *task_desc);

// Execute a  task.
int cods_exec_task(task_descriptor *task_desc);

// Execute a bundle
int cods_exec_bundle(bundle_descriptor *bundle_desc);

// Block for the completion of  task/bundle execution.
cods_status cods_wait_task_completion(task_descriptor *desc);
cods_status cods_wait_bundle_completion(bundle_descriptor *desc);

// Check the status of task/bundle execution.
cods_status cods_task_status(task_descriptor *desc);
cods_status cods_bundle_status(bundle_descriptor *desc);
```

Listing 5.1 presents the C programming interface for executing workflow tasks.

- *cods_add_task()* adds a task (described by *task_desc*) to a bundle. The function is used to construct a bundle of concurrently executing tasks. *task_desc* contains the basic task information, including task name, list of data variables that the task depends on. *task_desc* also specifies the resource requirement for the task execution, such as the number of required task executors.

- *cods_exec_task()/cods_exec_bundle()* submits the task/bundle execution request to workflow manager. The function is non-blocking and returns immediately after successful submission.

- *cods_wait_task_completion()/cods_wait_bundle_completion()* blocks for the completion of a submitted task/bundle.

- *cods_task_status()/cods_bundle_status()* returns the execution status of a submitted task/bundle. The function is non-blocking and returns immediately.

Programmer needs to write a driver program to implement workflow execution logic and

Listing 5.2: Example driver program of coupled XGC1-XGCa workflow.

```
void workflow_driver()
{
    cods_init();
    int i, num_coupling_step = 50;
    for (i = 1; i <= num_coupling_step; i++) {
        task_descriptor *xgc1, *xgca;

        /* Set up task descriptors.
           Detailed source code is omitted. */

        /* Execute xgc1 task */
        cods_exec_task(xgc1);
        cods_wait_task_completion(xgc1);

        /* Execute xgca task */
        cods_exec_task(xgca);
        cods_wait_task_completion(xgca);
    }
    cods_finalize();
}
```

the actual task graph. This section presents the programming examples of two representative workflows, including XGC1-XGCa workflow for plasma fusion science and DNS-LES workflow for combustion science (described in Chapter 2.1.2).

Listing 5.2 presents the example driver program of XGC1-XGCa workflow. The workflow has 2 tasks *xgc1* and *xgca*. The two tasks are executed in sequential order, and for multiple coupling steps. The driver program uses a outer loop to control the coupling steps. At each coupling step, the driver program first executes *xgc1* and waits for its completion, and then starts executing *xgca*. The driver program waits for the completion of *xgca* before advancing to the next coupling step.

Listing 5.3 presents the example driver program of DNS-LES workflow. The workflow has 2 tasks namely *dns* and *les*. *dns* and *les* executes concurrently in lockstep for multiple time steps. At each time step, data is transferred from *dns* to *les*. In this example, the driver program creates a bundle to include both tasks, executes the bundle and waits for its completion. The driver program does not need to explicitly control the time stepping using a control loop, because the time stepping is managed by *dns* and *les* program itself.

Listing 5.3: Example driver program of tightly coupled DNS-LES workflow.

```
void workflow_driver()
{
    cods_init();
    bundle_descriptor *b1;
    task_descriptor *dns, *les;

    /* Set up task descriptors.
       Detailed source code is omitted. */

    /* Execute dns and les tasks */
    cods_add_task(b1, dns);
    cods_add_task(b1, les);
    cods_exec_bundle(b1);
    cods_wait_bundle_completion(b1);
    cods_finalize();
}
```

Listing 5.4: Task placement C programming interface.

```
// Retrieve architectural information of task executors.
compute_resource* cods_get_compute_resource_info();

// Divide task executors on the allocated compute nodes
// into programmer-defined partitions.
int cods_build_partitions(compute_resource *info);

// Set location hint for a task.
int cods_set_location_hint(task_descriptor *task_desc,
                      unsigned char partition_type);

// Set data hint for a task.
int cods_set_data_hint(task_descriptor *task_desc,
                      data_hint *hint);
```

5.5.2 Task placement

This section presents the task placement APIs (see Listing 5.4) that allows programmer to explicitly specify task placement hint.

- *cods_get_compute_resource_info()* retrieves the architectural information of all task executors from workflow manager. *compute_resource* data field *executor_tab* uses a table to record the architectural information, such as compute node id and network topology, for each task executor.

- *cods_build_partitions()* defines customized functional partitions over the task executors. Programmer-defined partition type (a integer value in current implementation)

Listing 5.5: S3D data analytics workflow: example driver program.

```
enum programmer_defined_partition_type {
    SIMULATION = 1,
    INTRANSIT
};

void workflow_driver()
{
    cods_init();
    task_descriptor *s3d;
    compute_resource *info;

    info = cods_get_compute_resource_info();
    /* Update info->executor_tab[i].partition with
       programmer-defined partition type.
       Detailed source code is omitted. */

    /* Build partitions */
    cods_build_partitions(info);

    /* Set up task descriptors.
        Detailed source code is omitted. */

    /* Execute s3d task */
    cods_set_location_hint(s3d, SIMULATION);
    cods_exec_task(s3d);
    cods_wait_task_completion(s3d);
    cods_finalize();
}
```

can be assigned to executor entry in table *executor_tab*. For example, one typical customization for online data analytics workflow is to divide task executors into simulation and in-transit partitions. By default, task executors are not associated with any partition, and programmer-provided location hint has no effect.

- *cods_set_location_hint()*/*cods_set_data_hint()* sets placement hint for a task.

This section uses S3D data analytics workflow to demonstrate the use of task placement APIs. The workflow executes a simulation task *s3d* as the data producer, and three analysis tasks as the data consumers, including *viz* (visualization), *stat* (descriptive statistics) and *topo* (topological analysis). Simulation task *s3d* runs for multiple time steps, and the data analysis is performed in every n time steps. Value of n determines the frequency of online data analysis. In the presented example, the value of n is defined as 1. Programming of the example workflow involves two parts, which is described as follow.

Listing 5.6: S3D data analytics workflow: code snippets for s3d task program.

```
void s3d_execute_analysis()
{
    task_descriptor *stat, *viz, *topo;
    /* Set up task descriptors.
       Detailed source code is omitted. */

    /* Execute analysis tasks in-transit */
    cods_set_location_hint(stat, INTRANSIT);
    cods_set_location_hint(viz, INTRANSIT);
    cods_set_location_hint(topo, INTRANSIT);
    cods_exec_task(stat);
    cods_exec_task(viz);
    cods_exec_task(topo);
}

int s3d()
{
    int ts;
    for (ts = 1; ts <= num_time_step; ts++) {
        /* Computation.
             Detailed source code is omitted. */

        /* Execute analysis tasks. */
        if (rank == 0) s3d_execute_analysis();
    }
}
```

Listing 5.5 presents the driver program, which is used to bootstrap the workflow execution by submitting *s3d* task. Similar to DNS and LES solvers, *s3d* task program manages the time stepping by itself, and the driver program does not need a control loop to explicitly manages the *s3d* time stepping. In addition, the driver program retrieves the computation resource information, and divides the task executors into SIMULATION and INTRANSIT partitions by calling *cods_build_partitions()*. Driver program sets the location hint of *s3d* task as SIMULATION, and submits the task execution request.

Listing 5.6 presents the analysis execution logic, which is directly embedded into *s3d* task program. At each time step, *s3d* rank 0 process invokes routine *s3d_execute_analysis()* to start the asynchronous execution of data analysis tasks. As shown in Listing 5.6, location hint is specified and used to control the placement. In this specific example, all the three data analysis tasks are preferred to execute on INTRANSIT partition.

5.6 Applications

This section presents using CoDS task execution framework to support two real world coupled simulation workflows.

5.6.1 Online feature-based object tracking for scientific simulation data

Overview

In order to extract insightful information from large datasets produced by simulations over thousands of time steps, scientists often need to follow data objects of interest (i.e., features) across the different time steps. For example, meteorologists track storm formation and movement in climate modeling simulation while physicists identify burning regions in combustion simulations. As a result, feature extraction and tracking is an important technique for analyzing and visualizing scientific datasets. However, most feature extraction and tracking techniques operate offline by post-processing data written into files by the simulation runs. Being able to perform such feature-based analytics in-situ, i.e., concurrently with a simulation itself, can significantly improve the utility of these techniques at large scale. It can also lead to better utilization of expensive high-end resources as well as the overall productivity of the simulations.

This section presents the online in-situ feature-based object tracking on time-varying simulation data. The application is a 3D computational fluid dynamics (CFD) simulation that contains evolving amorphous regions. The feature-based object tracking operation is implemented using DOC - a scalable decentralized and online clustering technique [59], [60], which executes in-situ and analyzes the simulation data while it is being generated. Connected voxel regions of interest – features – needs to be identified at each time step and tracked over multiple time steps in order to visualize the time-varying datasets. More specifically, we focus on tracking objects as thresholded connected voxel regions that evolve both in location and shape over time.

Figure 5.7 illustrates the placement of simulation processes and DOC workers across the allocated compute nodes. The prototype implementation builds on CoDS task execution framework to enable the co-located execution of simulation and analysis operations, where

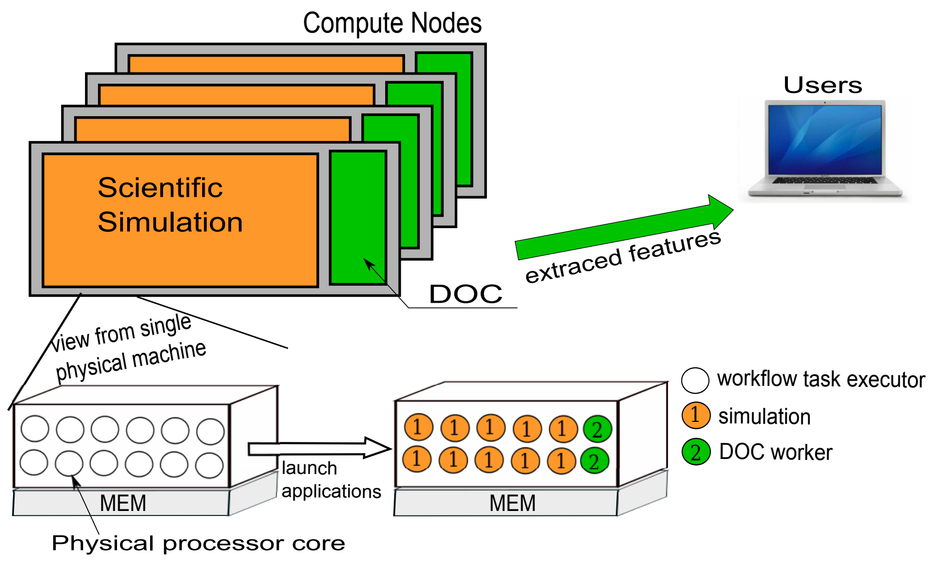


Figure 5.7: Architecture of the in-situ feature extraction and tracking system.

the physical processor cores on the multi-core compute nodes are functionally partitioned. DIMES (Chapter 4) is used to support sharing data between simulation and DOC through the node-local shared memory.

The prototype implementation was evaluated on the Lonestar linux cluster at Texas Advanced Computing Center (TACC). The Lonestar has 1,888 compute nodes, and each compute node contains two hex-core Intel Xeon processors, 24GB of memory and a QDR InfiniBand switch fabric that interconnects the nodes through a fat-tree topology. The system also supports a 1PB Lustre parallel file system.

Our evaluation consists of two parts. The first part evaluates the end-to-end data transfer performance of our in-situ data analysis framework, and also compares it with the traditional disk I/O approach. The second part evaluates the effectiveness and accuracy of our DOC-based feature tracking algorithm, using a time-varying dataset generated by simulation of coherent turbulent vortex structures.

Performance of data transfer

This section evaluates the end-to-end data transfer performance, and more specifically the time used to transfer data from simulation processes to DOC workers, for both our in-situ memory-to-memory and the disk I/O approaches. In this case, we use a testing MPI program as the parallel data producing simulation, which runs on a set of m processor cores. The parallel DOC workers runs on a separate set of n processor cores where the ratio of m:n is 8. In our in-situ data analysis approach, each DOC worker runs on a processor core co-located with 8 simulation cores of the same compute node, and retrieves data generated by the intra-node simulation processes. In our framework, the m:n ratio can be configured by users. From our experience, the simulation part of the experiment is usually more compute intensive, thus it makes sense to allocate a large amount of cores to simulation tasks. On the other hand, the data analysis part (DOC in this case) often requires a smaller number of cores. Our in-situ approach requires simulation and analysis tasks that exchange data to be placed on the same node in order to minimize data transfer and to reduce the overhead on the simulation itself. In the disk I/O approach, simulation processes dump data to disk with the one file per process method using binary POSIX I/O operations. Data files are then read by parallel DOC workers. For this evaluation, the number of simulation processes m is varied from 256 to 4,096, and the total data size produced by simulation at each timestep is varied from 8GB to 128GB. The testing program is configured to run for 100 timesteps at each data output size.

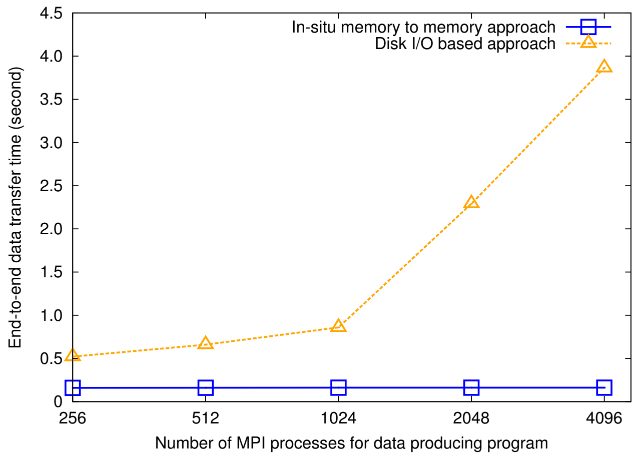


Figure 5.8: End-to-end data transfer time. The size of data produced per simulation process at each timestep is 32MB.

Figure 5.8 and 5.9 compare the performance of the two data transfer approaches. As shown in Figure 5.8, our in-situ memory-to-memory method is much faster than the disk I/O approach, with average speedup of transfer performance as about 10. Also, the in-situ memory-to-memory method is scalable, and shows no performance degradation when the

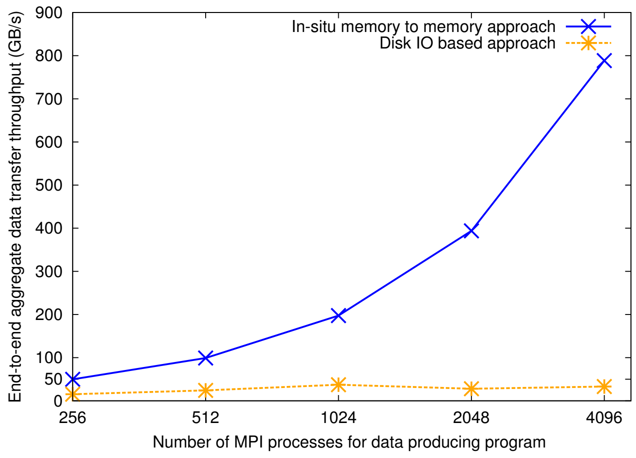


Figure 5.9: Aggregate data transfer throughput. The size of data produced per simulation process at each timestep is 32MB.

number of MPI processes in data producing program increases from 256 to 4096. The reason for this significant performance gain is that all data movement is intra-node, and performed through the on-node fast I/O path - shared memory. But for the disk I/O approach, both data producer and DOC worker processes have to use the off-node slow path - disk. Figure 5.9 illustrates the performance gain from another dimension - aggregate data transfer throughput. The fast intra-node shared memory approach enables much higher aggregate bandwidth for the data movement between simulation and DOC.

Effectiveness of the feature-based cluster tracking algorithm

This section evaluates the effectiveness and accuracy of our proposed feature tracking algorithm, using a time-varying 3D dataset. The dataset is generated by simulation of coherent turbulent vortex structures with 128^3 resolution (vorticity magnitude) and 100 time steps. In this case, the data cluster or object of interest is defined as thresholded connected voxel regions. These regions evolve both in location and shape during the simulation. Although different time steps of the dataset can be visualized offline using visualization tools such as Visit, it is difficult to visually observe and accurately follow regions of interest. The tracking information from our algorithm is used to determine how the regions evolve, e.g., size, location, density, over the time steps.

In this experiment, we define the regions of interest as the data points with vorticity values in the range of 9 to maximum. Since the tracked objects in our experiment vary fast

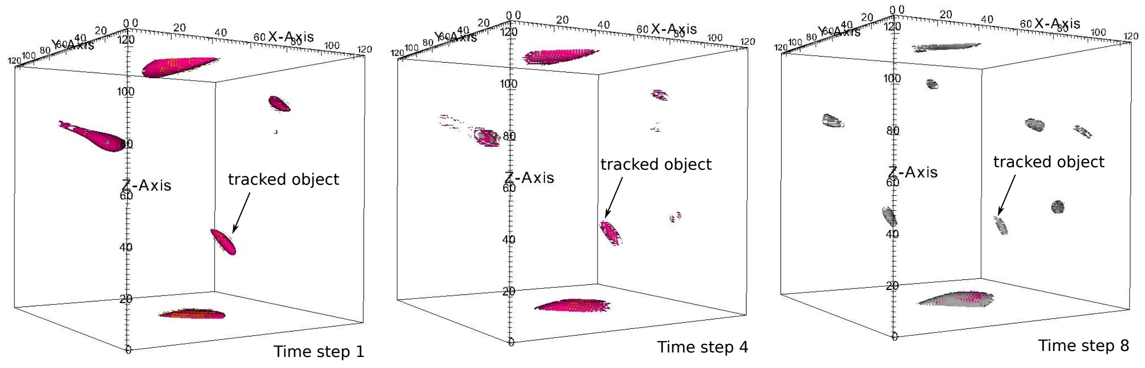


Figure 5.10: A visualized view of the evolving volume regions (objects) tracked by our feature-based tracking algorithm.

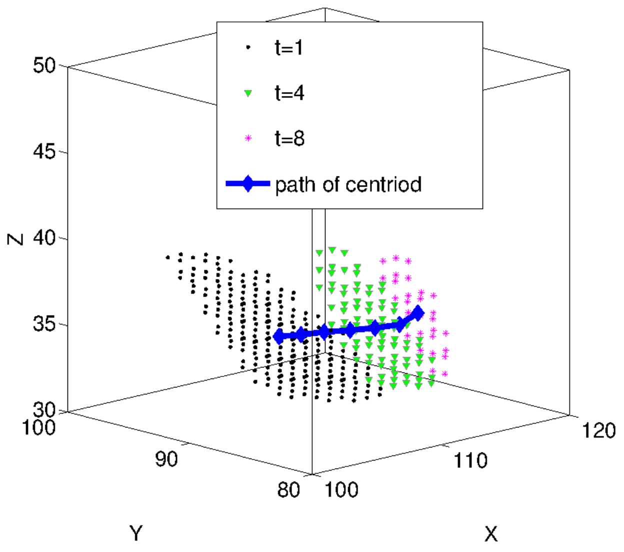


Figure 5.11: Illustration of tracked path for object evolves over multiple time steps.

and last only around 10 time steps, we snapshot three time steps of the visualized dataset for the duration of the tracked object, as shown in Figure 5.10. Also, it demonstrates the effective tracking of the evolving volume region (or object as in DOC) pointed by black arrows. We define *tracking accuracy* as the ratio of data points encompassed by the tracked objects to total number of points in the experiment. This ensures that high accuracy in object tracking can only be achieved by identifying paths from which the object pass through. In each experiment 50 frames were used to identify the paths of the objects within these 50 frames.

The tracking accuracy across 47 tests was 92.28% on average, meaning only 7.72% of all vortex points were not associated to any trackable object in our experiments. In Figure 5.11 we present the tracking of a object (the same one as in visualized Figure 5.10), as seen by DOC, at three different time steps. As shown in the figure, this object moves from left to right and shrinks in size.

5.6.2 Online data analytics for S3D combustion simulation

Overview

This section presents the online data analysis workflow for combustion simulation, which is implemented using CoDS task execution framework. The workflow couples parallel data analysis operations with S3D combustion simulation to analyze the raw simulation data on-the-fly, and decouple analysis operations from filed-based I/O by utilizing memory-based data sharing.

Figure 5.12 presents an overview of the S3D online data analysis framework. Task executor resource is divided into two distinct partitions namely SIMULATION and INTRANSIT. Task executors of SIMULATION partition is used for executing the S3D simulation, while the INTRANSIT partition is used for executing online data analysis operations. Data analysis tasks of the workflow is dynamically submitted and spawned by the S3D simulation, and workflow manager is responsible for the task scheduling and placement. DIMES is used to support asynchronous RDMA-based memory-to-memory data sharing between simulation and analysis tasks.

The online data analysis framework integrates two specific analysis operations including

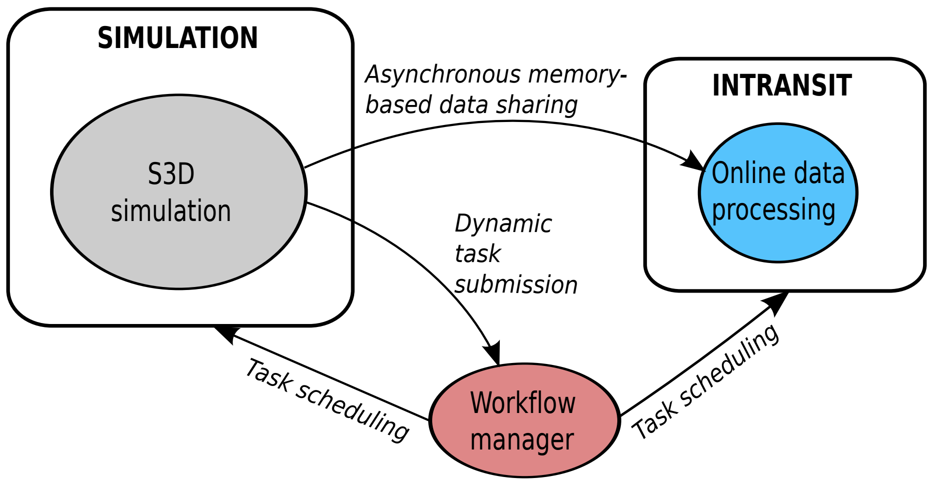


Figure 5.12: An overview of the online data analysis framework for S3D combustion simulation.

visualization, and value-based indexing.

- **Visualization**: Visualization enables scientists to visually monitor the simulations output. Our implementation uses the parallel volume rendering software library developed by UC Davis [70], which renders full-resolution simulation data. The parallel rendering approach generates high-quality images that can visually convey simulation results with great details.

- **Value-based indexing**: Value-based indexing enables scientists to formulate value-based range queries on the simulation data, and analyzes the data in an explorative manner. Our implementation uses software library FastBit [67] to generates value-based index of S3D simulation data. FastBit implements efficient compressed bitmap index technique, and has been demonstrated in a number of scientific applications to support value-based indexing and query processing.

Results

The prototype implementation was evaluated on the Cray XK7 Titan system at Oak Ridge National Laboratory. Titan has 18,688 compute nodes, and each compute node has a 16-core AMD Opteron processor, 32GB memory, and a Gemini router that interconnects the nodes via a fast network with a 3D torus topology.

S3D simulation periodically outputs chemical species data - 21 3D double arrays. One double array that represents the combustion temperature is the input for visualization operation, while the other 20 double arrays are the input for value-based indexing operation. Each simulation process computes on a 3D $20 \times 20 \times 20$ grid, and generates about 1.28MB data for the 21 double arrays. The experiments vary the total number of S3D simulation processes from 64 to 16384, in order to evaluate the scalability and performance results at different scales.

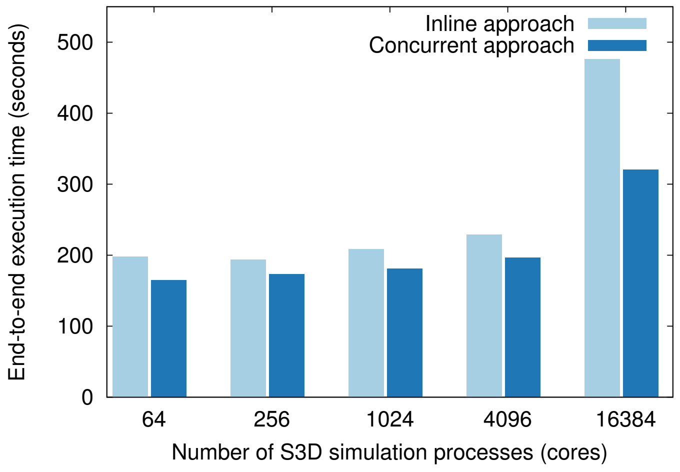


Figure 5.13: End-to-end execution time for S3D data analysis workflow.

In our experiments, S3D simulation runs for 50 time steps. Visualization and value-based indexing operation is performed at each time step. To demonstrate the benefits of **concurrent** data analysis supported by our framework, we compare with the **inline** data analysis approach which performs analysis operations directly on the physical processor cores that execute the S3D simulation.

We measure the total end-to-end wallclock execution time as perceived by the simulation, and Figure 5.13 presents the evaluation results of both **concurrent** and **inline** approach. Concurrent approach reduces the end-to-end execution time by 16.7%, 10.6%, 13.1%, 14.1%, 32.7% respectively, when the number of S3D simulation cores is varied from 64 to 16384. The advantage of concurrent approach becomes more significant at larger scale. Because the analysis operations need to write results to files, the inefficient file-based I/O at larger scale causes significant overhead to simulation execution in the inline approach. In the concurrent approach, data analysis operations are offloaded to separate processor cores and performed asynchronously, which introduces minimum overhead to simulation execution.

Concurrent approach requires additional computation resource to perform data analysis operations, as compared with the **inline** approach. Table 5.1 presents the amount of additional computation resource used in concurrent approach, including the number of processor cores for running workflow manager and *information space* servers, and the number of processor corse for executing data analysis operations. The resource overhead is less than 2% for large scale experiments with 1024, 4096 and 16384 S3D simulation cores, which is acceptable considering the effective reduction of end-to-end execution time and support of online data analytics workflow.

Num. of S3D simulation cores	Num. of cores (workflow manager and information space servers)	Num. of cores (data analysis operations)	Overhead
64	1	1	3.12%
256	2	4	2.34%
1024	4	16	1.95%
4096	8	64	1.75%
16384	16	256	1.66%

Table 5.1: Computation resource used in the concurrent approach of S3D online data analysis.

5.7 Related work

5.7.1 Programming model for workflow composition

Traditionally, programming scientific workflow is based on the abstraction of acyclic task dependency graph. Workflow is described as a directed acyclic graph (DAG), where each graph vertex represents a task and the edge identifies the data dependency. A task becomes runnable only after all its parent tasks complete their executions. Scientific workflow management systems DAGMan [3], Pegasus [32], Makelow [12] are examples using the task dependency graph model. Task dependency graph enforces sequential execution order between parent and child tasks connected by a edge. Task execution is completion-dependent, which depends on the completion of parent tasks. However, the programming model does not support expressing data-driven execution where a task becomes runnable once its data dependencies are satisfied. Data-driven execution is required by many coupled simulation workflow scenarios, such as the S3D online analytics workflow described in Chapter 2.1.1.

Many recent scientific workflow programming systems employ the dataflow programming model. A DAG of tasks can be used to represent a dataflow, where each graph vertex represents a task and the edge represents data flow between tasks. PreDatA [73] provides the map-shuffle-reduce model for users to compose customized data processing pipeline, which essentially composes a fixed dataflow. Twister [39] provides a iterative MapReduce [31] programming model. Users can express scientific workflow as iterative execution of MapReduce tasks, which essentially composes a iterative dataflow. Swift [64, 15, 65] is a dataflow language designed for programming many task computing (MTC) workflow, and supports dataflow-driven task-parallel execution. Swift supports arbitrary dataflow, and the dataflow specification is from the dynamic evaluation of programs written in the concurrent, expressive Swift scripting language.

5.7.2 Software framework for online data analytics workflow

This section presents software frameworks that support online data analytics on large-scale HPC clusters. Online data analytics workflow is one representative scenario of coupled simulation workflow, which emerges to address the increasing performance gap between computation and disk I/O. The workflow couples simulation code with data analysis, and analyze the raw simulation data while it is being generated. Data analysis operations can execute *in-situ* on the same compute nodes that run the simulation code, or *in-transit* on a set of separate and dedicated compute nodes.

In-situ data analysis: Functional partitioning (FP) [52] runtime allocates dedicated processor cores, namely *helper cores*, on the same compute nodes that run the simulation, and uses the helper cores to perform specific data processing tasks, e.g. checkpointing, de-duplication, and data format transformation. Similarly, Damaris/Viz [37] framework utilizes helper cores to perform online in-situ visualization. Darmaris/Viz implements a shared memory, zero-copy communication model to support efficient data sharing between simulation and visualization tasks. GoldRush [74] framework executes in-situ data analysis by utilizing idle CPU cycles of the processor cores running the simulation. Unlike the helper cores approach, data analysis tasks in GoldRush share the same computation cores with simulation. GoldRush employs fine-grained scheduling to “steal” idle resources from simulation, with the objective of minimizing interference between the simulation and data analytics. In-situ data analysis reduces the size of network data movement by placing analysis tasks closer to simulation data. However, on-node resource sharing, e.g. CPU, memory, and network, may cause interference between simulation and analysis.

In-transit data analysis on staging nodes: GLEAN [63] framework supports in-transit data analysis on dedicated staging nodes. Simulation running on compute nodes can use GLEAN transparently through a standard I/O library such as pNetCDF and HDF5, which provides non-intrusive integration with existing applications. PreDatA [73] middleware offloads output data from a running simulation to dedicated staging nodes using asynchronous data extraction, and performs data pre-processing and analysis in-transit. PreDatA supports in-transit execution of user-defined operations, and provides users with a map-shuffle-reduce programming paradigm to build customized data stream processing in the staging area. In-transit data analysis minimizes the impact on simulation, by offloading analysis computation to separate staging nodes.

Hybrid online data analysis: Several recent work provides the flexibility of analysis placement, and supports both in-situ and in-transit data analysis. JITStager [9] enables users to apply customized data operators to simulation output data along the entire I/O pipeline. JITStager implements SmartTap to execute data operations in-situ on processor cores that run the simulation code. In addition, JITStager builds on software DataStager [10] to extract data from simulation to staging area, execute the data customization operators, and forward the data to downstream data analysis framework, i.e. PreDatA middleware, for further processing. FlexIO [75] middleware enables users to tune analytics placement to improve performance for online data analytics workflow. FlexIO can automatically configure the underlying transport to support placement decision made by users, and support various placement options, e.g. in-situ helper cores, in-transit staging nodes.

5.7.3 Task placement based on programmer-provided information

Several recent parallel programming languages and systems allow programmers to explicitly control the location of task execution or express "hint" that can be utilized by runtime to make the placement decision. PGAS programming language Chapel [25] provides the abstraction of *locale* to represent a portion of the target parallel machines that has computation and storage capabilities. For example, on HPC cluster of multicore nodes, each node can be abstracted as a *locale*. Programmer can use the Chapel *on* statement to control on which locale to execute a block of code. Similarly, PGAS language X10 [26] provides the abstraction of *place*, and programmer can use the X10 *at* statement to express the *place* of computation task. Parallel programming model Piccolo [58] builds application around kernel functions, and the application data is stored in in-memory tables that partitioned and distributed on different compute nodes. Piccolo allows programmer to express locality policies (or preferences) to co-locate a kernel execution with desired table partition. Piccolo runtime executes a kernel instance on machine that stores most of the required data, which minimizes remote reads.

5.8 Summary

This chapter presents CoDS framework that supports the programming, task execution and placement for coupled simulation workflows. CoDS provides the programming support by implementing asynchronous and dynamic task execution APIs. Programmers can compose workflow by combining concurrent, sequential and iterative executions of task programs. CoDS implements hint-based task placement mechanism that utilizes programmer-provided hint as input. Programmers can provide data hint for the locality-aware placement, and explicitly control the location of task execution by specifying location hint. This chapter also presents the experimental evaluation of the runtime framework by using several representative workflow applications.

Chapter 6

Communication- and topology-aware task mapping

6.1 Introduction

High-performance computing (HPC) systems are increasingly being based on two architectural trends to achieve better performance: First, systems have increased core count/density on a single compute node. Second, systems have increased number of compute nodes, which are interconnected by network with complex topology. For example, petascale supercomputer Cray XK7 Titan uses a 3D torus network to interconnect its 18,688 16-core compute nodes, and IBM BlueGene/Q Mira uses a 5D torus network to interconnect its 49,152 18-core compute nodes. This architectural trend makes the largest supercomputers highly hierarchical, and the cost of data communication can vary significantly depending on where the communicating processes are placed. For example, on-node data movement is more efficient than off-node data movement that requires transferring data over interconnection network. Data communication between compute nodes that are nearby in the network topology is more efficient than between compute nodes that are connected long-hop away. Meanwhile, emerging coupled scientific simulation workflows consist of multiple applications that need to interact and exchange significant volume of data at runtime. The cost of moving the increasingly large volumes of data associated with these interactions and couplings has become a dominant part of the overall workflow execution.

Clearly, in order to effectively utilize the potential of current and emerging HPC systems it is critical that such data-intensive coupled simulation workflows exploit *data locality* to the extent possible, increase the size of on-node data exchange and reduce the cost of data movement over network. To achieve this objective, one key research direction is task mapping, which determines the mapping of application processes onto network topology, and has significant impact on the cost of data movement. While existing research work [69]

[18] [44] [33] has focused on mapping frequently communicating processes within a single application onto processor cores that are physically “close”, mapping processes of multiple applications that are part of a workflow has not been studied.

This chapter presents the communication- and topology-aware task mapping to localize communication for coupled simulation workflow. First, the task mapping improves data locality by placing intensively communicating processes onto the same multi-core compute node, with the objective of reducing the size of data movement over interconnection network. Intra-node data movement can use more efficient shared memory, bypassing the network devices. Figure 6.1 illustrates the co-located execution of processes from two concurrently coupled applications. Second, the task mapping improves data locality by placing communicating processes onto physically “close” compute nodes in the network topology, with the objective of reducing the number of hops for data transfers over network.

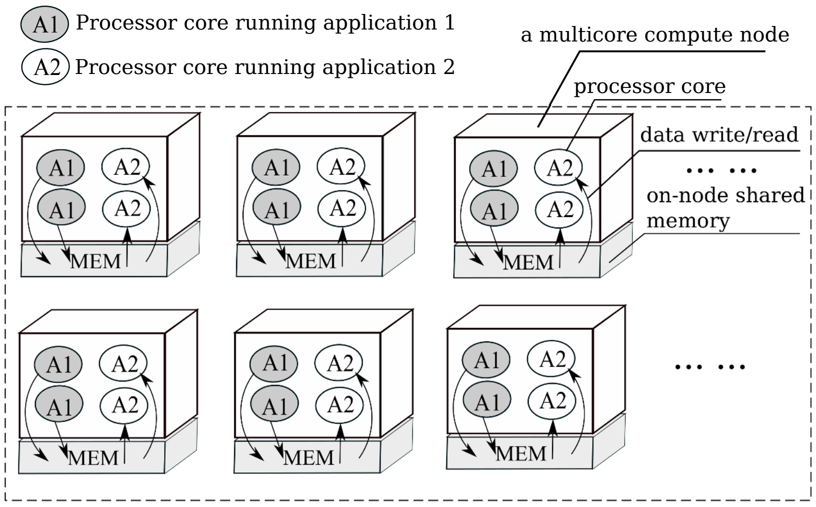


Figure 6.1: Illustration of localized data communication through co-located execution applications processes.

Unlike existing research work that focus on optimizing intra-application communication for single application, our method targets coupled simulation workflow that consists of multiple applications, and optimizes the overall workflow communication performance by considering both intra-application and inter-application communications. Moreover, while existing approaches focus on communication between application processes that execute simultaneously, our method also targets communication between sequentially executed applications, which is unique to coupled simulation workflow.

Communication- and topology-aware task mapping utilizes existing graph partitioning and topology mapping algorithms as the main building blocks. The prototype implementation builds on DIMES (presented in Chapter 4) to support memory-based inter-application data exchange, and builds on CoDS (presented in Chapter 5) to execute the workflow and control the runtime placement of application processes.

The remainder of the chapter is organized as follows. Section 6.2 formulates the task mapping problem. Section 6.3 presents algorithm description of the communication- and topology-aware task mapping method. Section 6.4 presents the integration of the task mapping method into CoDS framework. Section 6.5 presents the experimental evaluation using three representative testing workflows. Section 6.6 presents related research work and Section 6.7 summarizes this chapter.

6.2 Task mapping problem

This section first formulates the task mapping problem using the task graph based workflow representation (described in Chapter 5.2). In addition, this section defines the performance metrics, and describes the assumptions we made for the task mapping problem.

6.2.1 Formalization

- **Workflow communication graph**: Each workflow application is a parallel program that executes on a massive number of processes. Workflow communication includes intra-application communication which moves data between processes in the same application, and inter-application communication which moves data between different applications. Workflow communication is represented by a weighted, undirected graph $G_c = (V_c, E_c, w_c)$. V_c is the set of application processes in a workflow, and E_c defines the set of graph edges. For each edge $(u, v) \in E_c$, $w_c(u, v)$ represents the size of communication between application process u and v.

- **Network topology graph**: The physical interconnection network is represented by a undirected graph $G_t = (V_t, E_t, c_t)$. V_t is the set of physical nodes (compute node or network switch) in the network topology, and an edge $(u, v) \in E_t$ represents a direct network link between node u and v. Capacity value $c_t(u)$ defines the number of application processes that can execute on compute node $u \in V_t$. The value of $c_t(u)$ is 0 for switches (if any in the network topology), and is equal to the number of processor cores on a compute node. For example, the value $c_t(u)$ is set as 16 for Cray XK7 Titan supercomputer where each compute node has 16 processor cores.

- **Task mapping**: The task mapping is represented by a function

$$\tau : V_c \rightarrow V_t \tag{6.1}$$

which maps the vertices of V_c (workflow application processes) to the vertices of V_t (physical compute nodes). As shown by previous research work [11] [44], the task mapping problem can be formulated as mapping the communication graph to network topology graph, which is essentially a graph embedding problem and is known to be NP-hard.

6.2.2 Quality measures

- **Size of network data movement** measures the amount of inter-node data communication. On high-end computing system that consists of multi-/many-core compute nodes, communication can be generally categorized as two types: intra-node communication and inter-node communication. Intra-node communication is performed using much faster, efficient on-node memory. Inter-node communication needs to transfer data over network, which could have higher overhead in terms of latency and energy.

- **Hop-bytes** [11] [66] measures the total size of data communication in bytes weighted by network distance. Network distance is measured as the length (number of network hops) of the shortest path between communicating processes in the network topology graph. Given the communication graph G_c, network topology graph G_t, and a specific task mapping τ, the hop-bytes $HB(G_c, G_t, \tau)$ can be computed as

$$HB(G_c, G_t, \tau) = \sum_{(u,v)\in E_c} w_c(u, v).d(\tau(u), \tau(v)) \tag{6.2}$$

where $d(\tau(u), \tau(v))$ denotes the network distance between compute node $\tau(u)$ and

$\tau(v)$. For application processes running on the same compute node, the network distance is defined as 0.

- **Communication time** measures the total communication time as perceived by the application during the workflow execution. Reduction of communication time is an important indicator of the task mapping quality, because reducing the communication time directly reduces the end-to-end workflow execution time.

6.2.3 Assumptions

One assumption we made is that the communication graph does not change across different iterations of workflow execution, and the mapping of application processes to compute nodes does not need to change at runtime. As a result, the communication- and topology-aware task mapping is static and only needs to be computed once before the workflow execution. In practice, this assumption is valid for most coupled simulation workflows we have been working on. Task mapping for workflows exhibiting dynamic communication patterns is beyond the scope of current research work, and will be explored in future work.

6.3 Communication- and topology-aware task mapping

This section starts with an overview of the communication- and topology-aware task mapping method, and then describes the task mapping algorithm in details.

6.3.1 Overview

Existing topology mapping methods for single application typically involve two key steps. First, inter-process communication graph of the application is partitioned into N/n equal pieces, where N is the total number of application processes and n denotes the number of processes that can execute on a compute node. With the initial partitioning, the inter-process communication graph is reduced to a inter-partition communication graph. Second, inter-partition communication graph is mapped to the network topology graph, which determines the mapping of application processes to networked compute nodes. Topology task mapping is a well-known NP-hard problem, and many heuristic algorithms have been proposed, including greedy approaches [11] [44], recursive bipartitioning [44] [66], graph similarity [44], and geometric based mapping methods [17] [33].

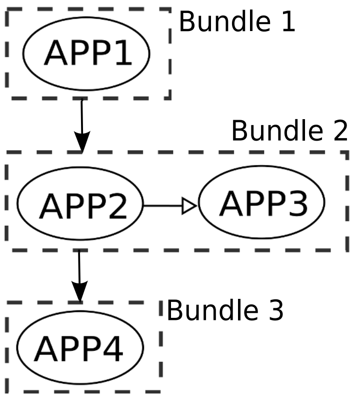


Figure 6.2: Illustration of task graph that can be represented as pipeline of application bundles.

Our communication- and topology-aware task mapping for coupled simulation workflow builds on graph partitioning and topology mapping algorithms used by existing mapping methods. However, our method differentiates from single application task mapping, and aims at reducing the cost of data movement caused by both intra-application and inter-application communication. In many coupled simulation workflows, applications can be represented as a pipeline of bundles, as illustrated by Figure 6.2. Our task mapping method optimizes the data communication between concurrently coupled applications in the same bundle, such as APP2 and APP3 in bundle 2. In addition, our task mapping method also optimizes the data communication between sequentially coupled applications from different bundles, such as APP1 and APP2.

The communication- and topology-aware workflow task mapping involves three key steps:

- Partition the communication graph for each application bundle, in order to reduce the size of network data movement within a bundle, which includes intra-application communication and inter-application communication.

- Group the partitions to reduce the size of network data movement between sequential bundles, which includes inter-application communication between sequentially coupled applications.

- Map the groups onto the physical compute nodes.

6.3.2 Method description

In this section, we will use a synthetic workflow example to demonstrate the step-by-step execution for the task mapping. Figure 6.3 shows the DAG representation of the workflow example, which consist of three applications. The workflow applications APP1, APP2 and APP3 run on 12, 8 and 4 processes respectively, and share a common data domain represented by a 2D cartesian grid. The workflow executes bundle 1 and bundle 2 in sequential order, which starts executing APP2 and APP3 after the completion of APP1. Figure 6.3 also shows the decomposition and distribution of the data domain over application processes. In this example, we define the number of processor cores on a compute node as 6.

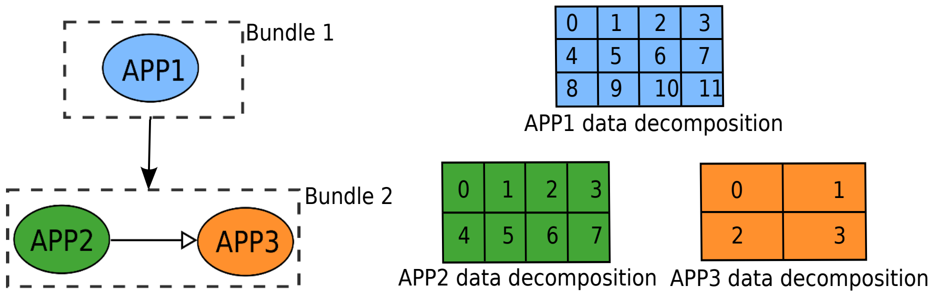


Figure 6.3: Simple workflow example used to demonstrate the communication- and topology-aware task mapping: (Left) DAG representation and (Right) the applications data decomposition.

The **first step** is to divide application processes into partitions, and the objective is to minimize the size of network data movement between application processes in the same bundle. Algorithm used in this step aims at mapping heavily communicating processes to the same partition. Application processes mapped to the same partition would execute on the same compute node and exchange data using shared memory.

Input of the partitioning is the bundle communication graphs $G_{b1}, G_{b2}, ...G_{bn}$, where $V_c = V_{b1} \cup V_{b2} \cup ... \cup V_{bn}$ (V_c is the set of all application processes in the workflow). Output of the partitioning is the mapping from application processes to a set of partitions:

$$\tau_1 : V_c \rightarrow V_p, \tag{6.3}$$

where V_p is the set of partitions. The size of a partition equals to the core count of a compute node. Graph partitioning is applied to each bundle, and our method uses graph partitioning library METIS [47] which offers optimized partitioning heuristics. Figure 6.4 illustrates the step 1 for mapping the example workflow.

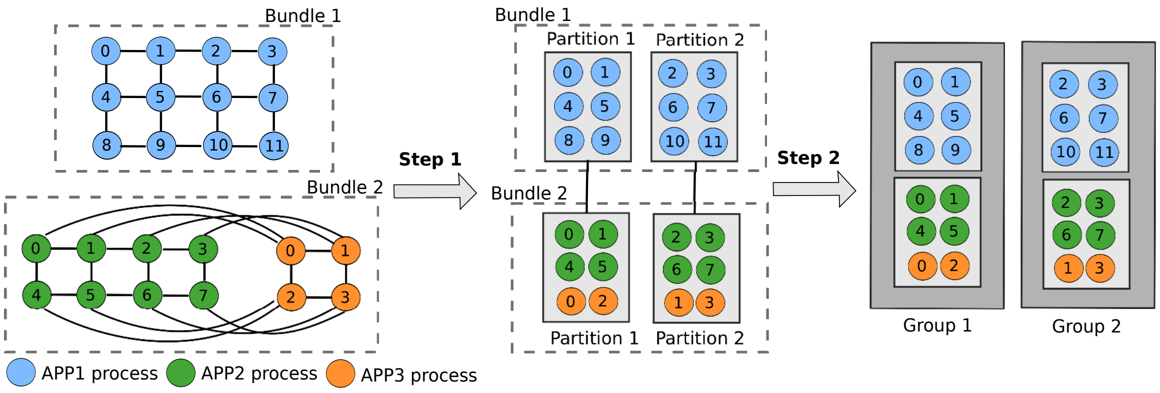


Figure 6.4: Illustrations: (1) Partitioning of communication graph - application processes in each bundle is divided into partitions to localize the data communication. (2) Grouping of the graph partitions - map data consumer processes to where the input data is generated to increase the amount of data reuse from local group

The **second step** is to assign partitions into groups, and the objective is to minimize the size of network data movement between sequentially executed bundles. Algorithm used in this step aims at mapping heavily communicating partitions (from sequential bundles) to the same group, which essentially places application processes closer to input data. For example, as presented in Figure 6.4, Bundle1-Partition1 and Bundle2-Partition1 are mapped to group 1, Bundle1-Partition2 and Bundle2-Partition2 are mapped to group 2. Data sharing between partitions of the same group can also utilize shared memory, which is enabled by the local in-memory staging capability of DIMES (presented in Chapter 4). Data sharing between the sequentially executed bundles is decoupled in time. For example, data generated by APP1 needs to be available after the completion of APP1, which is read by APP2 and APP3. The traditional approach to support this decoupled-in-time data sharing is through disk files. DIMES caches data generated by the producer application in the local memory of compute nodes. Consumer applications can directly read the data from local memory if appropriate task mapping is achieved.

Input of the grouping is the partitions computed in step 1, and the output is the mapping from partitions to groups:

$$\tau_2 : V_p \rightarrow V_g, \tag{6.4}$$

where V_g is the set of groups. The number of groups is equal to the total number of compute nodes required for the workflow execution. Figure 6.4 illustrates step 2 for mapping the example workflow. It builds a inter-partition communication graph to represent data movement between sequential bundles. Graph vertex represents a partition computed in step 1, and graph edge represents the amount of data communication between partitions. Similar to the previous step, graph partitioning technique is applied to this inter-partition communication graph, to divide the partitions into groups.

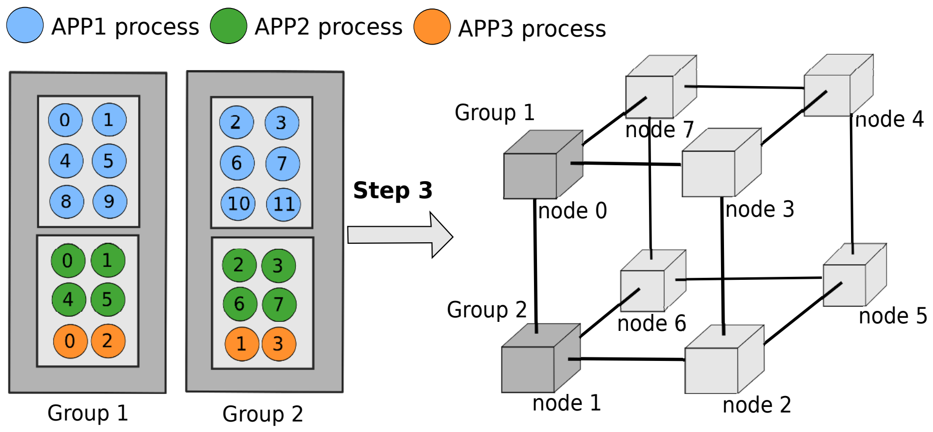


Figure 6.5: Topology mapping: map the groups onto compute nodes interconnected by the underlying network.

The **last step** performs mapping of groups onto the topology of underlying interconnection network, and the objective is to minimize the network hop-bytes. After the first two steps, the mapping from application processes to groups ($V_c \rightarrow V_g$) can be acquired. The original inter-process communication graph can be reduced to a inter-group communication graph. Input of the topology mapping is the inter-group communication graph G_g and network topology graph G_t, the output is the mapping from groups to compute nodes:

$$\tau_3 : V_g \rightarrow V_t, \tag{6.5}$$

Figure 6.5 uses a 3D $2 \times 2 \times 2$ mesh network to illustrate the mapping of the sample workflow onto the compute nodes. The principle is to map communicating groups to nearby compute nodes to reduce the network distance of data transfers. In this case, the groups are mapped to two adjacent compute nodes, node 0 and 1, in the network topology graph.

Algorithm 1 Recursive bisection mapping recursive_map()

```
Input: Communication graph $G_g$, network graph $G_t$
Output: Mapping $\tau : V_g \rightarrow V_t$
1: if $|V_t| == 1$ then
2:     $\tau(u) = v$, for $u \in V_g$, $v \in V_t$;
3:     return;
4: end if
5: $(G_{g1}, G_{g2}) \leftarrow$ bipartition($G_g$);
6: $(G_{t1}, G_{t2}) \leftarrow$ bipartition($G_t$);
7: if $|V_{g1}| == |V_{t1}|$ then
8:     recursive_map($G_{g1}$,$G_{t1}$);
9:     recursive_map($G_{g2}$,$G_{t2}$);
10: else
11:     recursive_map($G_{g1}$,$G_{t2}$);
12:     recursive_map($G_{g2}$,$G_{t1}$);
13: end if
```

Topology mapping applies the recursive bisection based algorithm [44] [66], as presented in Algorithm 1. In this method, the weighted communication graph G_g is recursively split with minimum edge-cut into equal halves, while the topology graph G_t is split with maximum edge-cut. The runtime complexity of this algorithm is $O(|E_g|log(|V_g|)+|E_t|log(|V_t|))$.

After the completion of above three steps, the mapping $\tau : V_c \rightarrow V_t$ (defined in Equation 6.1) can be derived from Equation 6.3, 6.4 and 6.5. As a result, workflow application process are successfully mapped to compute nodes in the network topology. For compute node with multi-core processors, another level of mapping is required within the physical node to map each application process to a specific processor core. Our approach employs sequential mapping, which first orders the processes mapped to the same node by MPI rank, and then sequentially assigns the processes to physical cores. Advanced node-local mapping, such as node architecture-aware approach [66] [75], can be utilized. But it is beyond the scope of our current research work and will be investigated in future work.

6.4 Implementation

The communication- and topology-aware task mapping method has been implemented and integrated into the CoDS framework (presented in Chapter 5). CoDS framework consists of four main components, including workflow manager, task executor, task submitter and information space. Task executors span across the allocated compute nodes. Each task executor runs on one physical processor core, and is used to execute one application process. Workflow manager is responsible for scheduling and allocating task executors for workflow execution.

In our prototype implementation, CoDS workflow manager performs communication- and topology-aware task mapping, and generates the mapping from application processes to cores. The complete execution flow is described as below.

- **Obtain network topology graph**: Workflow manager needs to obtain the network topology graph of the compute nodes used for workflow execution, in order to perform the task mapping. CoDS framework uses system tools or library to query the network topology information for compute nodes, and then build the network topology graph. For example, on Cray Gemini 3D-torus interconnection network, CoDS uses the Cray RCA library function *rca_get_meshcoord()* to retrieve network coordinate of a compute node.

- **Generate workflow communication graph**: Workflow manager also needs to obtain the workflow communication graph representing both intra-application and inter-application communications. In practice, two approaches can be used to generate the communication graph: (1) Communication graph can be inferred or estimated for applications presenting simple, regular and structured communications. For example, many scientific applications are based on multidimensional cartesian grid data domain, and the data decomposition can be expressed in terms of a domain size, process layout, data distribution type, and data block size. A n-tuple $(s_1, ..., s_n)$ and n-tuple $(p_1, ..., p_n)$ is used to specify the size and number of processes in each dimension of the data domain. As a result, the intra-application communication graph can be easily computed if the application processes exhibit regular communication pattern such as near-neighbor data exchange. Similarly, communication graph can be computed for regular inter-application communication pattern, such as $M \times N$ array redistribution. (2) Tracing tools can be used to extract inter-process communications, and generate more accurate communication graphs. However, most tracing tools are not scalable, which can only trace applications running at small scale.

- **Apply the task mapping method**: After obtaining the network topology and workflow communication graph, workflow manager executes the communication- and topology-aware task mapping algorithm. Because most application programs are implemented using MPI, output of this step is the mapping from application MPI ranks to task executors.

- **Runtime mapping of application processes**: CoDS framework uses a technique called "rank reordering" [21] [55] to achieve the actual placement of application processes to task executors. Task executors first dynamically create a new MPI communicator, and then reorder the MPI rank values according to the mapping result computed in previous step. Application program's data decomposition and inter-process communication depends on the reordered MPI rank values. As a result, the rank reordering successfully enforces the desired process placement.

6.5 Evaluation

This section uses three representative testing workflows to evaluate the effectiveness and scalability of the task mapping approach. The experimental evaluation is performed on Titan Cray XK7 supercomputer at Oak Ridge National Laboratory's National Center for Computational Sciences. Titan has 18,688 compute nodes, and each compute node has a 16-core processor and 32GB node-level memory. Titan uses Gemini, a 3D-torus network topology, to interconnect the compute nodes. The 3D torus network of Titan has a global dimension $25 \times 16 \times 24$, where each Gemini ASIC in the 3D torus network provides two network interface controllers (NICs) and can connect to two compute nodes.

6.5.1 Experiment setup

The application used in the experiment is a 3D Stencil benchmark in MPI. The application has a 3D global data domain, and organizes the application processes into a regular 3D grid. The global data domain is decomposed and distributed to the application processes according to the layout of the process grid. During the application execution, each process needs to communicate with six neighbors and exchange boundary information about local data domain. In this testing application, the intra-application data communication is mainly dominated by the near-neighbor data exchange. The inter-application data communication involves extracting data for the entire domain from M processes of one application to N processes of another application, which is known as $M \times N$ data redistribution.

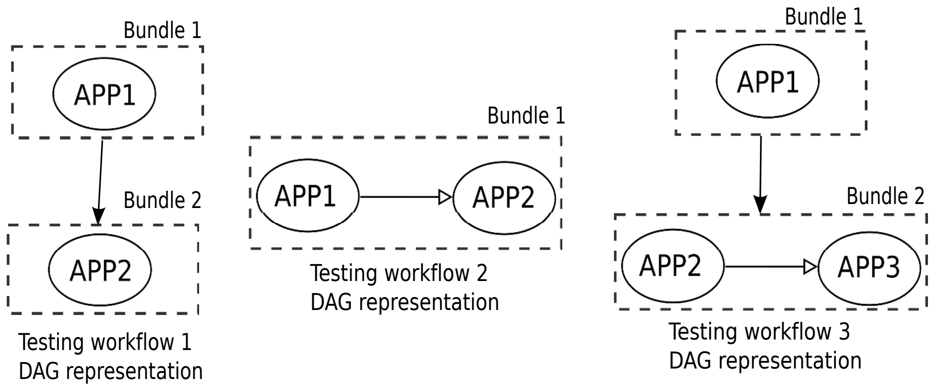


Figure 6.6: DAG representations for the testing workflows used in the experimental evaluation.

The experiment used three testing workflow scenarios that represent typical interaction patterns in coupled simulation workflows, as illustrated by Figure 6.6. The first testing workflow *wf1* consists of two applications *wf1-app1* and *wf1-app2*. The workflow executes the two applications in a sequential order, and data generated by *wf1-app1* needs to be redistributed to processes of *wf1-app2*. The second testing workflow *wf2* also consists of two applications, namely *wf2-app1* and *wf2-app2*, but the two applications are executed concurrently and data is redistributed from application *wf2-app1* to *wf2-app2*. The third testing workflow *wf3* has a more complex interaction patterns and consists of three applications. The workflow first executes application *wf3-app1*, and then launches two concurrently executing applications *wf3-app2* and *wf3-app3* after the completion of *wf3-app1*. As shown in Figure 6.6, the workflow has two different interaction patterns, firstly sharing data from *wf3-app1* to *wf3-app2* and *wf3-app3*, secondly sharing data between *wf3-app2* and *wf3-app3*.

Table 6.1 to 6.3 presents the setup of application global data domain, number of processes for each application. For each testing workflow, the experiment configures 4 different cases to increase the total number of processor cores from about 4K to 32K, in order to evaluate the scalability of our mapping approach.

	Global data domain	*wf1-app1* number of process	*wf1-app2* number of process
Case-1	512×1024×1024	4096	1024
Case-2	1024×1024×1024	8192	2048
Case-3	1024×1024×2048	16384	4096
Case-4	1024×2048×2048	32768	8192

Table 6.1: Testing workflow 1 - configurations of global data domain and the number of application processes

	Global data domain	*wf2-app1* number of process	*wf2-app2* number of process
Case-1	512×1024×1024	4096	256
Case-2	1024×1024×1024	8129	1024
Case-3	1024×1024×2048	16384	2048
Case-4	1024×2048×2048	32768	4096

Table 6.2: Testing workflow 2 - configurations of global data domain and the number of application processes

	Global data domain	*wf3-app1* number of process	*wf3-app2* number of process	*wf3-app3* number of process
Case-1	512×1024×1024	4096	2048	512
Case-2	1024×1024×1024	8192	4096	1024
Case-3	1024×1024×2048	16386	8192	2048
Case-4	1024×2048×2048	32768	16384	4096

Table 6.3: Testing workflow 3 - configurations of global data domain and the number of application processes

Our experimental evaluation compare communication- and topology-aware task mapping with the sequential mapping approach. Sequential mapping is the default process-to-core mapping approach used by the job scheduler of most HPC clusters, including ORNL Titan, which serves as the baseline approach in our evaluation. Sequential mapping approach places the application processes onto the compute nodes in a sequential way purely based upon the process MPI rank, and is oblivious to either the intra-application or inter-application communication patterns.

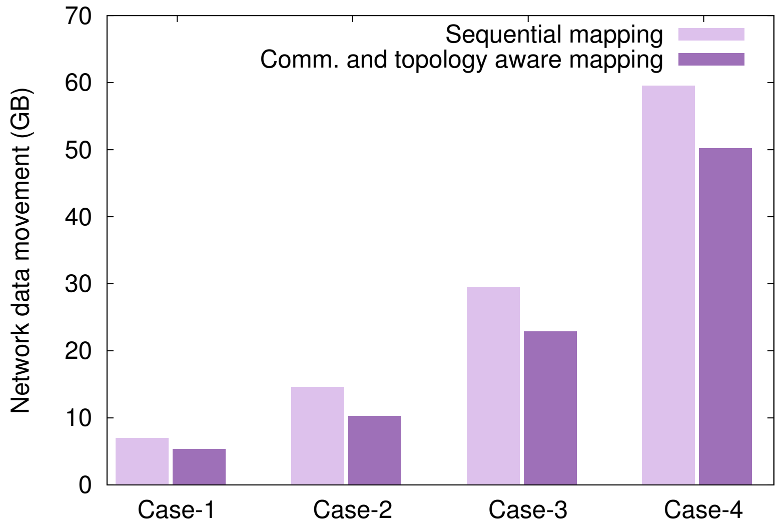
6.5.2 Size of data movement over network

This section measures the reduction in the size of data movement over network. Figure 6.7(a) to 6.7(c) presents the evaluation results of total network data transfers. As shown in the figures, the communication- and topology-aware task mapping approach effectively reduces the size of network data communication, for all the three testing workflows. Compared with sequential mapping, the communication and topology-aware task mapping approach in testing workflow 1 reduces the the amount of network data transfer by 24% (1.69 GB), 29% (4.46 GB), 22% (6.74 GB), 15% (9.55 GB) respectively for the different configured cases. For testing workflow 2, the communication and topology-aware task mapping reduces the network data transfer by 10% (0.66 GB), 11% (1.49 GB), 11% (3.08 GB), 12% (6.82 GB). For testing workflow 3, the communication and topology-aware task mapping reduces the network data transfer by 29% (4.74 GB), 34% (11.73 GB), 27% (18.72 GB), 20% (26.49 GB).

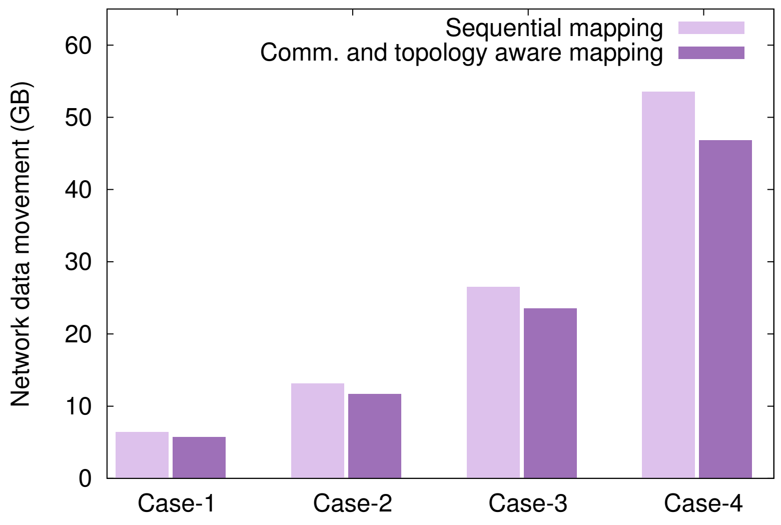
6.5.3 Hop-bytes

Figure 6.8(a) to 6.8(b) presents the evaluation results of total network hop-bytes. For testing workflow 1, compared with sequential mapping, the communication and topology-aware task mapping approach reduces the network hop-bytes by 60%, 71%, 61%, 53% respectively for the different configured cases. For testing workflow 2, our approach reduces network hop-bytes by 48%, 47%, 44%, 52%. For testing workflow 3, our approach reduces the network hop-bytes by 54%, 61%, 53%, 46%.

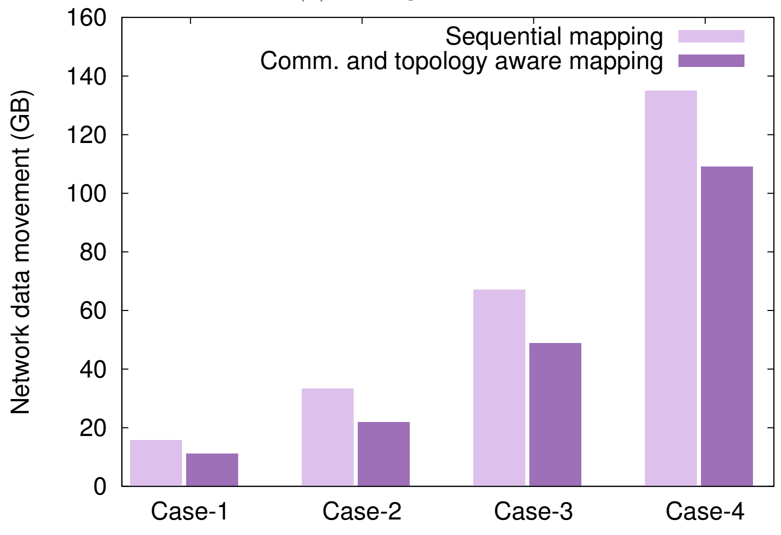
Figures 6.9(a) to 6.9(c) presents the reduction of network hops for transferred data



(a) Testing workflow 1

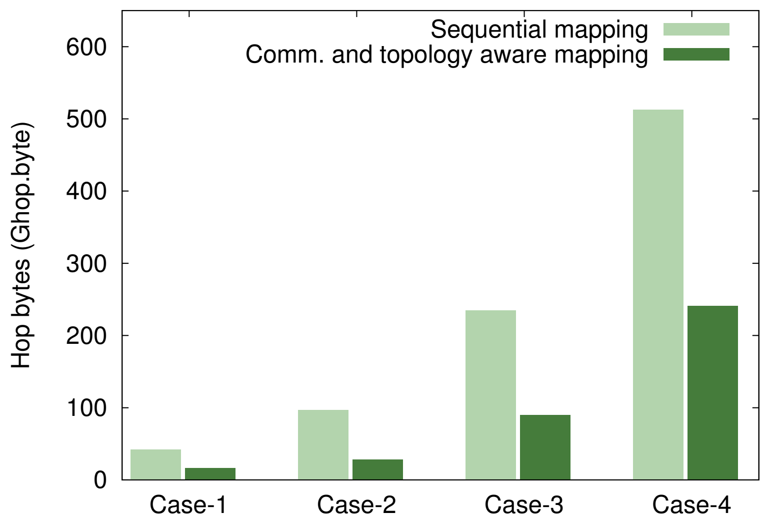


(b) Testing workflow 2

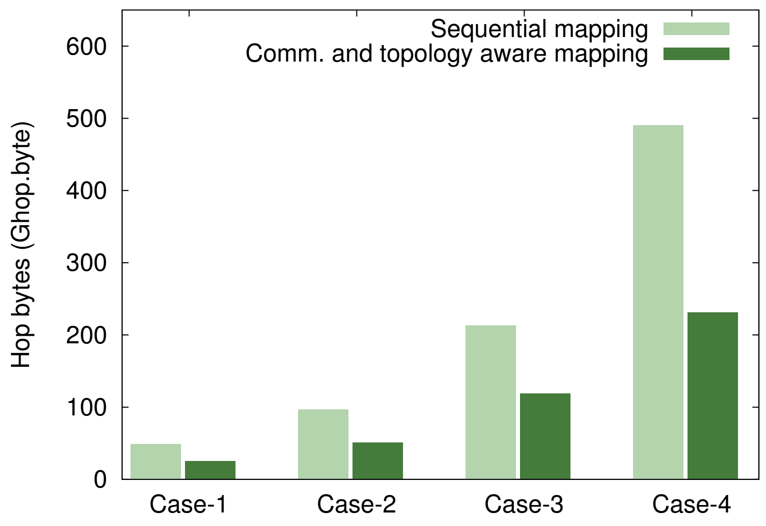


(c) Testing workflow 3

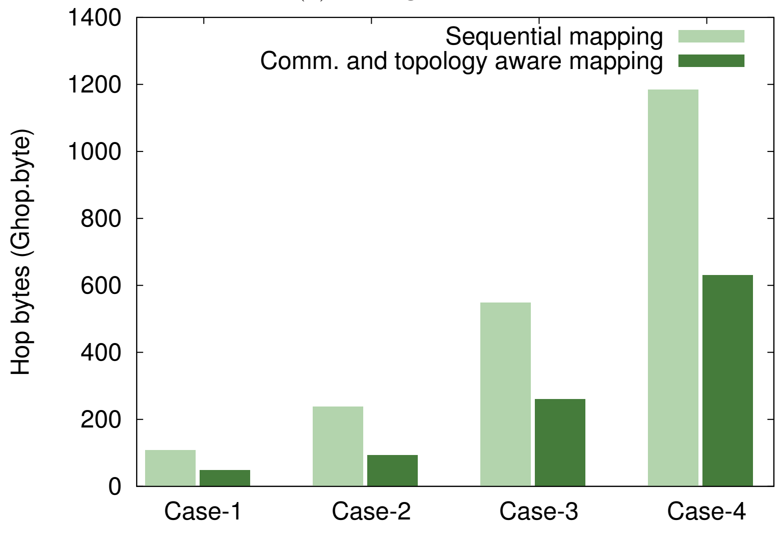
Figure 6.7: Compare the size of network data communication.



(a) Testing workflow 1

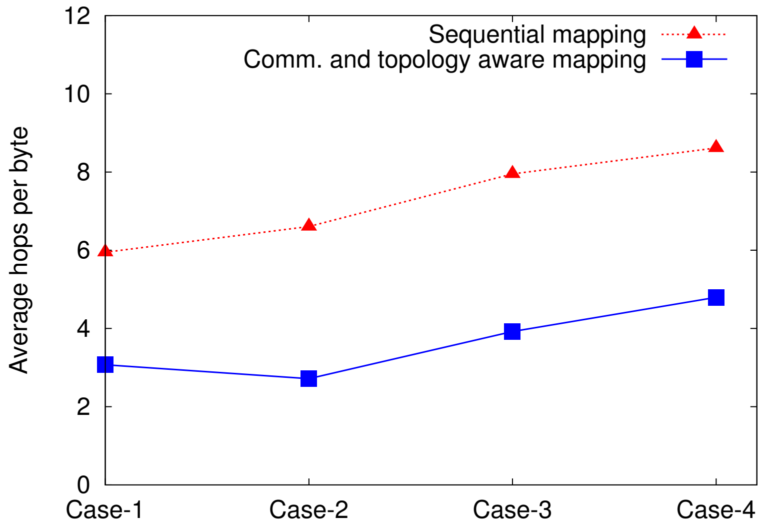


(b) Testing workflow 2

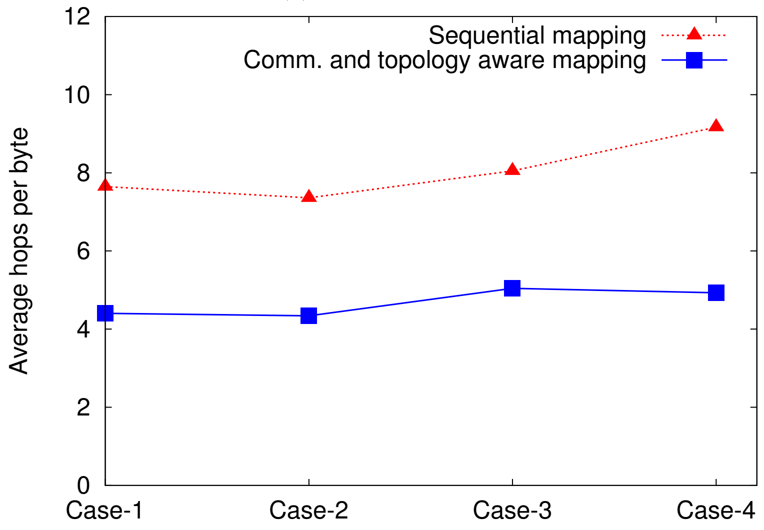


(c) Testing workflow 3

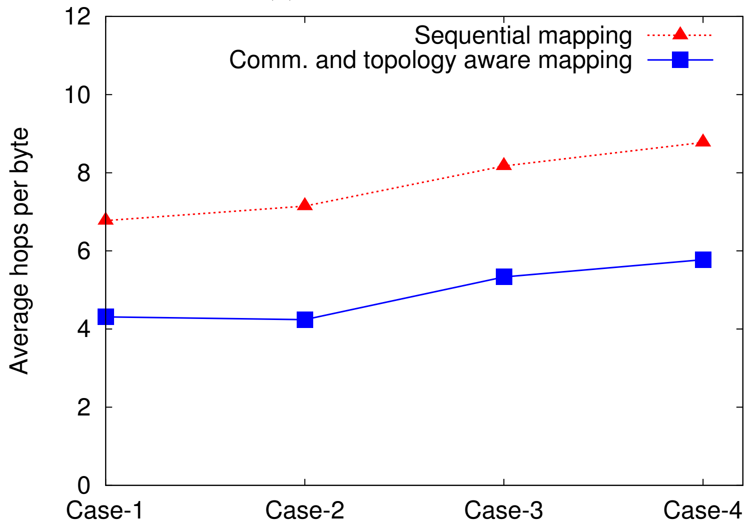
Figure 6.8: Compare the network hop-bytes.



(a) Testing workflow 1



(b) Testing workflow 2



(c) Testing workflow 3

Figure 6.9: Compare the average hop per byte.

using another performance metric - average hop per byte. This metric shows in average how many network hops each data byte would transfer. As shown by the figures, our communication- and topology-aware task mapping approach also effectively reduces the average hop/byte for all the three testing workflows. In addition, it can be observed from Figure 6.9 that the task mapping has more significant reduction in average hop per byte for experimental configurations that use larger number of processor cores, such as Case-3 and Case-4 which uses about 16K and 32K. This is because the diameter or maximum number of network hops of the underlying network topology graph increases, inappropriate placement of the application processes could potentially cause longer network distance for the data communications.

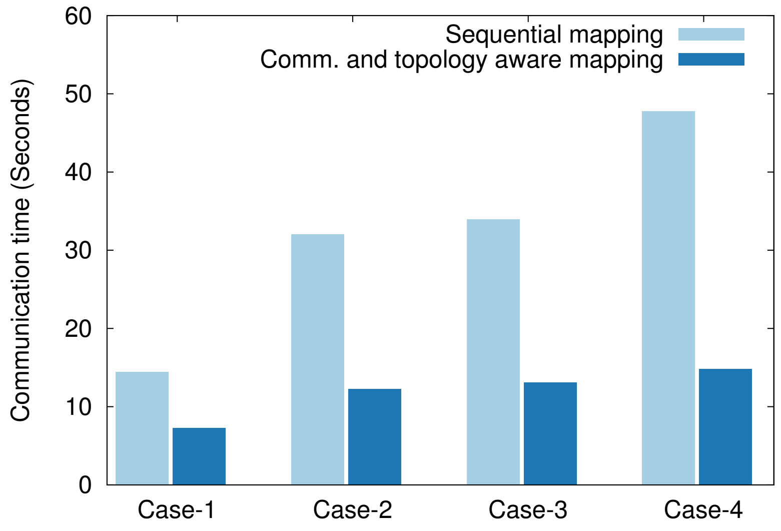
6.5.4 Communication time

This experiment evaluates the reduction in communication time. Each testing workflow is executed for 100 iterative runs, and the total communication time is measured and presented. The measured communication activities include the intra-application near-neighbor data exchanges, and the inter-application data communication.

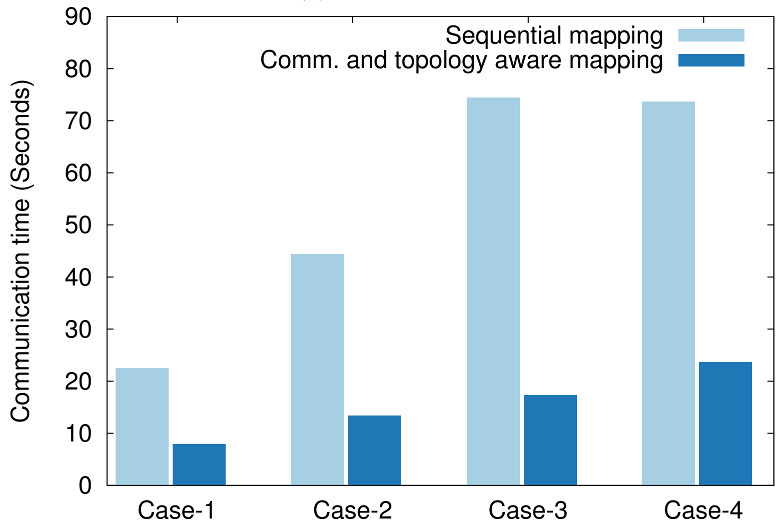
As shown by Figure 6.10(a) to 6.10(c), the communication- and topology-aware task mapping approach effective reduces the communication time for all the three testing workflows, when compared with the default mapping approach. For testing workflow 1, the total communication time is reduced by about 49% to 69%. For testing workflow 2, the total communication time is reduced by about 64% to 76%. For testing workflow 3, the total communication time is reduced by about 54% to 68%.

6.6 Related work

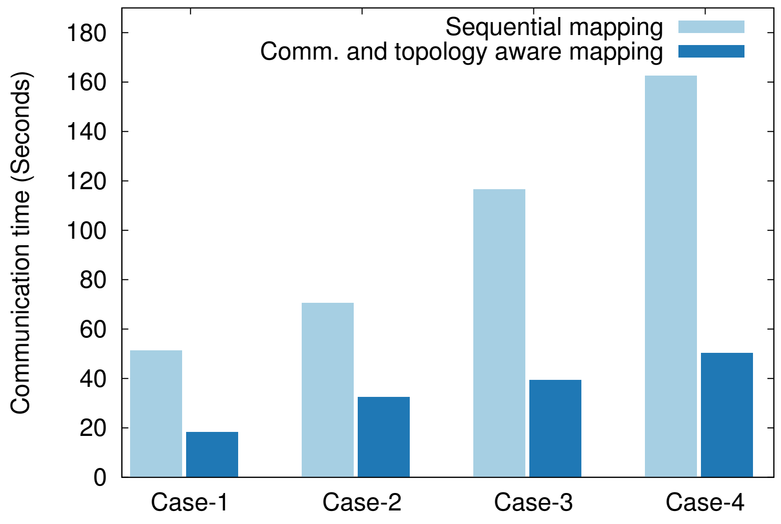
Mapping application processes onto parallel computers has been well studied. Bokhari [20] reduces the mapping problem to graph isomorphism, and develops a heuristic algorithm that starts with a initial mapping followed by sequences of pairwise interchanges. Lee and Aggarwal [51] proposes similar two-stage optimization approach with initial greedy assignment and pairwise swaps. With the advent of parallel computers that interconnect tens of



(a) Testing workflow 1



(b) Testing workflow 2



(c) Testing workflow 3

Figure 6.10: Compare the data communication time.

thousands of multi-core compute nodes, recent research work focuses on developing mapping heuristics that can be applied to the high-performance computing systems. Yu, Chung, and Moreira [69] proposes different topology mapping strategies to map regular Cartesian processes structures onto the torus network of BlueGene/L supercomputer. Bhatelei [18] presents a automated mapping framework that uses a suite of heuristic techniques to map regular communication patterns to 2D and 3D processor meshes, on both BlueGene/P and Cray XT supercomputers. Hoefler and Snir presents LibTopoMap [44], a topology mapper library that supports task mapping for generic (both regular and irregular) application communication graphs and network topology graphs. Most existing research work targets at mapping single parallel application, our approach and framework focuses on coupled simulation workflows consisting of both concurrently and sequentially executing applications, and considers both the intra-application and inter-application communications.

6.7 Summary

This chapter presents the communication- and topology-aware task mapping for coupled simulation workflow, which aims at reducing the size of data movement over network and the number of hops for data transfers over network. Specifically, the task mapping applies graph partitioning technique to workflow communication graph, in order to map intensively communication application processes onto the same multi-core compute node, so as to reduce the data movement over network. In addition, the approach applies topology mapping to reduce the *hop-bytes*. Experimental evaluation is performed on ORNL Titan, using three representative benchmark workflows. The evaluation results show effective reduction of the size of network data movement and end-to-end communication time.

Chapter 7

Conclusions and future work

Emerging coupled scientific simulation workflows are composed of multiple applications that need to interact and exchange data at runtime, which have the potential to achieve higher accuracy and accelerate the data to insight process. However, running data-intensive coupled simulation workflows on parallel machines with thousands of compute nodes is non-trivial. While existing HPC programming and runtime systems have focused on single application, many research challenges remain unsolved for coupled simulation workflow that couples and executes multiple interacting applications.

This thesis identifies and addresses key problems and requirements for coupled simulation workflow. Specifically, this thesis presents the programming interface and runtime mechanisms to support workflow composition and execution, in-memory data management, task placement. Firstly, this thesis presents DIMES data management framework to support memory-based data sharing and exchange between coupled applications. DIMES co-locates distributed in-memory staging on compute nodes that run the applications, and caches data in node-local memory. Data is indexed according to its spatial geometric domain, and applications can access data of interest using insert/retrieve queries that contains spatial constraints, e.g., Cartesian bounding box. Secondly, this thesis presents CoDS task execution framework to support the workflow composition, task execution and placement. In CoDS, programmers write a driver program to compose the workflow and orchestrate the execution of component applications through the task execution APIs. CoDS implements locality-aware task placement, and allows programmers to express locality preference for a task by providing *data hint*. In addition, CoDS allows users to programmatically divide the allocated computation resource into functional partitions, and explicitly express the placement affinity by providing *location hint*, e.g., the preferred functional partition for task execution. Finally, this thesis presents communication- and topology-aware task mapping to optimize the workflow communication performance. The thesis presents a holistic method to map workflow communication graph onto physical network topology graph, with the objective to improve data locality and localize communications. We evaluated the effectiveness, scalability and performance of the proposed programming interface and runtime mechanisms, through integration and experiments with several real-world coupled simulation workflows.

In the future, this work can be extended in several directions.

- **Utilizing new storage technologies**: Next-generation supercomputers are integrating new storage technologies, such as faster solid-state drives (SSD) and non-volatile random-access memory (NVRAM), to increase the node-local storage capacity. Current implementation of co-located data staging only utilizes DRAM as the node-local storage resource. This can be extended to include node-local SSD or NVRAM storage resources, which can increase the staging capacity and free more space of the precious DRAM for applications.

- **Handle dynamic runtime behaviors of coupled simulation workflow**: This thesis focuses on coupled simulation workflows that exhibit static regular application data distribution and inter-application communication. However, emerging workflows start to exhibit dynamic runtime behaviors. For example, scientific simulations based on adaptive mesh refinement (AMR) have the data distribution dynamically changing at runtime. Another example is dynamic workflow where the data interaction pattern between applications is not known in advance. At runtime, new applications may join an existing group of workflow applications and start data interaction in a dynamic on-demand fashion. This presents the new requirement for autonomic runtime data and analytics management, which can capture the dynamic runtime behavior and adapt the data and analytics placement.

References

[1] Com official site. https://www.microsoft.com/com/default.mspx. Accessed September 2014.

[2] Corba official site. http://www.corba.org/. Accessed September 2014.

[3] Dagman. http://research.cs.wisc.edu/htcondor/dagman/dagman.html. Accessed September 2014.

[4] Hierarchical data format, version 5. http://www.hdfgroup.org/HDF5/. Accessed September 2014.

[5] Lustre official site. http://wiki.lustre.org/index.php/Main_Page. Accessed September 2014.

[6] Nersc cori. https://www.nersc.gov/users/computational-systems/cori/. Accessed March 2015.

[7] Network common data form. http://www.unidata.ucar.edu/software/netcdf/. Accessed September 2014.

[8] Ornl summit. https://www.olcf.ornl.gov/summit/. Accessed March 2015.

[9] H. Abbasi, G. Eisenhauer, M. Wolf, K. Schwan, and S. Klasky. Just In Time: Adding Value to The IO Pipelines of High Performance Applications with JITStaging. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, June 2011.

[10] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng. Datastager: scalable data staging services for petascale applications. In *Proceedings of the 18th International Symposium on High Performance Distributed Computing*, June 2009.

[11] T. Agarwal, A. Sharma, A. Laxmikant, and L. Kale. Topology-aware Task Mapping for Reducing Communication Contention on Large Parallel Machines. In *Proceedings of 20th IEEE International Parallel and Distributed Processing Symposium*, April 2006.

[12] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: a portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, May 2012.

[13] N. Ali, P. Carns, K. Iskra, D. Kimpe, S. Lang, R. Latham, R. Ross, L. Ward, and P. Sadayappan. Scalable i/o forwarding framework for high-performance computing systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, September 2009.

[14] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of the 8th International Symposium on High Performance Distributed Computing*, August 1999.

[15] T. Armstrong, J. Wozniak, M.Wilde, K. Maheshwari, D. Katz, M. Ripeanu, E. Lusk, and I. Foster. ExM: High-level dataflow programming for extreme-scale systems. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Parallelism*, June 2012.

[16] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. W. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pbay, D. Thompson, H. Yu, F. Zhang, and J. Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *Proceedings of IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2012.

[17] A. Bhatele. Automating topology aware mapping for supercomputers. 2010.

[18] A. Bhatele and, G. Gupta, L. Kale and, and I.-H. Chung. Automated Mapping of Regular Communication Graphs on Mesh Interconnects. In *Proceedings of the International Conference on High Performance Computing*, December 2010.

[19] J. Biddiscombe, J. Soumagne, G. Oger, D. Guibert, and J.-G. Piccinali. Parallel computational steering for hpc applications using hdf5 files in distributed shared memory. *IEEE Transactions on Visualization and Computer Graphics*, 18(6):852–864, June 2012.

[20] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, 30(3):207–214, Mar. 1981.

[21] B. Brandfass, T. Alrutz, and T. Gerhold. Rank reordering for mpi communication optimization. *Computers & Fluids*, 80:372–380, 2013.

[22] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to upc and language specification. *Technical Report CCS-TR-99-157, George Washington University*, 1999.

[23] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur. Pvfs: A parallel file system for linux clusters. In *In Proceedings of the 4th Annual Linux Showcase and Conference*, October 2000.

[24] N. Carriero and D. Gelernter. Linda in context. *Communication of ACM*, 32(4):444–458, 1989.

[25] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[26] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, October 2005.

[27] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorski, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science and Discovery*, 2:1–31, 2009.

[28] W. D. Collins, C. M. Bitz, M. L. Blackmon, G. B. Bonan, C. S. Bretherton, J. A. Carton, P. Chang, S. C. Doney, J. J. Hack, T. B. Henderson, J. T. Kiehl, W. G. Large, D. S. McKenna, B. D. Santer, and R. D. Smith. The Community Climate System Model Version 3 (CCSM3). *Journal of Climate*, 19(11):2122–2143, 2006.

[29] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki. Flexpath: Type-based publish/subscribe system for large-scale science analytics. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2014.

[30] E. F. D'Azevedo, J. Lang, P. H. Worley, S. A. Ethier, S.-H. Ku, and C. Chang. Hybrid mpi/openmp/gpu parallelization of xgc1 fusion simulation code. In *Proceedings of ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2013.

[31] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[32] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. B. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13:219–237, July 2005.

[33] M. Deveci, S. Rajamanickam, V. J. Leung, K. Pedretti, S. L. Olivier, D. P. Bunde, U. V. Catalyürek, and K. Devine. Exploiting geometric partitioning in task mapping for parallel computers. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium*, May 2014.

[34] C. Docan, M. Parashar, J. Cummings, and S. Klasky. Moving the Code to the Data - Dynamic Code Deployment Using ActiveSpaces. In *Proceedings of 25th IEEE International Parallel and Distributed Processing Symposium*, May 2011.

[35] C. Docan, M. Parashar, and S. Klasky. DataSpaces: An Interaction and Coordination Framework or Coupled Simulation Workflows. In *Proceedings of 19th International Symposium on High Performance and Distributed Computing*, June 2010.

[36] C. Docan, M. Parashar, and S. Klasky. Enabling high-speed asynchronous data extraction and transfer using dart. *Concurrency and Computation: Practice and Experience*, 22:1181–1204, 2010.

[37] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro. Damaris/viz: A nonintrusive, adaptable and user-friendly in situ visualization framework. In *Proceedings of IEEE Symposium on Large-Scale Data Analysis and Visualization*, October 2013.

[38] G. Edjlali, A. Sussman, and J. H. Saltz. Interoperability of Data Parallel Runtime Libraries. In *Proceedings of the 11th International Symposium on Parallel Processing*, April 1997.

[39] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, June 2010.

[40] P. Fasel and S. Mniszewski. PAWS: Collective Interactions and Data Transfers. In *Proceedings of the 10th International Symposium on High Performance Distributed Computing*, August 2001.

[41] D. Gelernter. Generative communication in linda. *ACM Transaction on Programming Language System*, 7(1):80–112, 1985.

[42] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[43] W. D. Gropp and E. L. Lusk. Dynamic process management in an mpi setting. In *Proceedings of 7th IEEE Symposium on Parallel and Distributed Processing*, April 1995.

[44] T. Hoefler and M. Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the International Conference on Supercomputing*, May 2011.

[45] M. A. Jette, A. B. Yoo, and M. Grondona. Slurm: Simple linux utility for resource management. In *Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP)*, June 2003.

[46] T. Jin, F. Zhang, Q. Sun, H. Bui, M. Parashar, H. Yu, S. Klasky, N. Podhorszki, and H. Abbasi. Using cross-layer adaptations for dynamic data management in large scale coupled scientific workflows. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2013.

[47] G. Karypis and V. Kumar. METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0. Technical report, Dept. of Computer Science, Univ. of Minnesota, 1995.

[48] S. Ku, C. Chang, and P. Diamond. Full-f gyrokinetic particle simulation of centrally heated global itg turbulence from magnetic axis to edge pedestal top in a realistic tokamak geometry. *Nuclear Fusion*, 49(11):115021, 2009.

[49] J. Larson, R. Jacob, and E. Ong. The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models. *International Journal of High Performance Computing Applications (IJHPCA)*, 19:277–292, August 2005.

[50] J.-Y. Lee and A. Sussman. High Performance Communication between Parallel Programs. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*, April 2005.

[51] S.-Y. Lee and J. K. Aggarwal. A mapping strategy for parallel processing. *IEEE Transactions on Computer*, 36(4):433–442, April 1987.

[52] M. Li, S. S. Vazhkudai, A. R. Butt, F. Meng, X. Ma, Y. Kim, C. Engelmann, and G. Shipman. Functional partitioning to optimize end-to-end performance on many-core architectures. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2010.

[53] Q. Liu, J. Logan, Y. Tian, H. Abbasi, N. Podhorszki, J. Y. Choi, S. Klasky, R. Tchoua, J. Lofstead, R. Oldfield, et al. Hello adios: the challenges and lessons of developing leadership class i/o frameworks. *Concurrency and Computation: Practice and Experience*, 26(7):1453–1473, 2014.

[54] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, Metadata Rich IO Methods for Portable High Performance IO. In *Proceedings of 23th IEEE International Parallel and Distributed Processing Symposium*, May 2009.

[55] G. Mercier and E. Jeannot. Improving mpi applications performance on multicore clusters with rank reordering. In *Recent Advances in the Message Passing Interface*, pages 39–49. Springer, 2011.

[56] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications (IJHPCA)*, 20(2):203–231, May 2006.

[57] R. Oldfield, P. Widener, A. Maccabe, L. Ward, and T. Kordenbrock. Efficient data movement for lightweight i/o. In *Proceedings of IEEE International Conference on Cluster Computing*, September 2006.

[58] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of 9th USENIX Symposium on Operating Systems Design and Implementation*, October 2010.

[59] A. Quiroz, N. Gnanasambandam, M. Parashar, and N. Sharma. Robust clustering analysis for the management of self-monitoring distributed systems. *Cluster Computing*, 12(1):73–85, Mar. 2009.

[60] A. Quiroz, M. Parashar, N. Gnanasambandam, and N. Sharma. Design and evaluation of decentralized online clustering. *ACM Transactions on Autonomous and Adaptive Systems*, 7(3):34:1–34:31, Oct. 2012.

[61] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation or the sun network filesystem, 1985.

[62] F. Schmuck and R. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *In Proceedings of USENIX Conference on File and Storage Technologies*, January 2002.

[63] V. Vishwanath, M. Hereld, V. A. Morozov, and M. E. Papka. Topology-aware data movement and staging for i/o acceleration on blue gene/p supercomputing systems. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2011.

[64] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[65] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/t: Large-scale application composition via distributed-memory dataflow processing. In *Proceedings of 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2013.

[66] J. Wu, Z. Lan, X. Xiong, N. Y. Gnedin, and A. V. Kravtsov. Hierarchical task mapping of cell-based amr cosmology simulations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, November 2012.

[67] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006.

[68] W. Xu, Y. Lu, Q. Li, E. Zhou, Z. Song, Y. Dong, W. Zhang, D. Wei, X. Zhang, H. Chen, et al. Hybrid hierarchy storage system in milkyway-2 supercomputer. *Frontiers of Computer Science*, 8(3):367–377, 2014.

[69] H. Yu, I.-H. Chung, and J. Moreira. Topology mapping for blue gene/l supercomputer. In *Proceedings of the ACM/IEEE Conference for High Performance Computing, Networking, Storage and Analysis*, November 2006.

[70] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K.-L. Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, 30(3):45–57, 2010.

[71] L. Zhang and M. Parashar. Enabling efficient and flexible coupling of parallel scientific applications. In *Proceedings of 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*, April 2006.

[72] F. Zheng, H. Abbasi, J. Cao, J. Dayal, K. Schwan, M. Wolf, S. Klasky, and N. Podhorszki. In-situ i/o processing: a case for location flexibility. In *Proceedings of the 6th Workshop on Parallel Data Storage*, November 2011.

[73] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf. PreDatA - preparatory data analytics on peta-scale machines. In *Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium*, April 2010.

[74] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky. Goldrush: resource efficient in situ scientific data analytics using fine-grained interference aware execution. In *Proceedings of ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2013.

[75] F. Zheng, H. Zou, G. Eisenhauer, K. Schwan, M. Wolf, J. Dayal, T.-A. Nguyen, J. Cao, H. Abbasi, S. Klasky, N. Podhorszki, and H. Yu. Flexio: I/o middleware for location-flexible scientific data analytics. In *Proceedings of 27th IEEE International Parallel and Distributed Processing Symposium*, May 2013.