In-Staging Data Placement for Asynchronous Coupling of Task-Based Scientific Workflows

Qian Sun*, Melissa Romanus*, Tong Jin*, Hongfeng Yu†,
Peer-Timo Bremer‡, Steve Petruzza¶, Scott Klasky§, Manish Parashar*
*Rutgers University, Piscataway, NJ 08854, USA, §Oak Ridge National Labortory, Oak Ridge, TN 37831, USA
†University of Nebraska-Lincoln, Lincoln, NE 68588, USA, ¶University of Utah, Salt Lake City, UT 84112, USA
‡Lawrence Livermore National Laboratory, Livermore, CA 94550, USA

***Abstract*—Coupled application workflows composed of applications implemented using task-based models present new coupling and data exchange challenges, due to the asynchronous interaction and coupling behaviors between tasks of the component applications. In this paper, we present an adaptive data placement approach that addresses these challenges by dynamically adjusting to the asynchronous coupling patterns. Specifically, it places data across a set of staging cores/nodes with an awareness of the application-specific data locality requirements and the runtime task executions at these staging cores/nodes, with the goal of reducing end-to-end execution time and data movement overhead of the workflow. We experimentally demonstrate the effectiveness of our approach on the Titan Cray XK7 system using representative data coupling patterns derived from current scientific workflows. The evaluation demonstrates that our approach efficiently improves performance by reducing the time-to-solution and increasing the quality of insights for scientific discovery.**

***Keywords*—Data storage systems, Couplings, Runtime**

I. INTRODUCTION

Advanced application workflows running on current and planned extreme scale systems have the potential to enable dramatic insights into complex phenomena in a range of disciplines. Recent research has shown that in-memory data staging and in-situ/in-transit data processing approaches (e.g., DataSpaces[6] / ActiveSpaces[7] and FlexPath[4]) can be used to address data related challenges resulting from the interactions and data exchange requirements of coupled application workflows executing at extreme scales. These approaches offload data to separate data staging resources, i.e., in-memory storage distributed across cores/nodes on the system where the workflow is running. They also use these staging resources to handle mismatches in data representation, data distribution and data production/consumption rates, and to support interactions and data exchanges between the coupled applications. Note that staging resources may also be co-located with application on the same set of compute nodes and utilize node-local storage resources to stage data and preserve data locality, as shown in Figure 1.

While these research efforts continue to address data related challenges of coupled application workflows on current systems, growing scales (and projected billion-way concurrency at exascale) are bringing challenges related to exploiting extreme

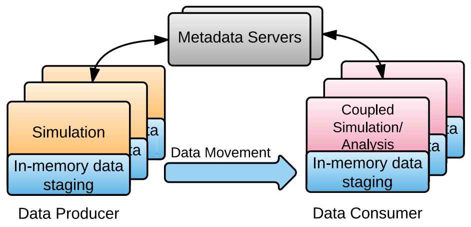
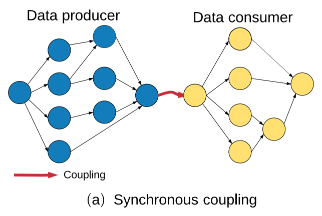


Fig. 1: A scientific workflow composed of coupled simulations and analysis components, implemented using data staging.

levels of parallelism to the forefront, and are leading to the adoption of task-based runtimes, such as Charm++ [2]. These runtimes employ an asynchronous execution model suited for extreme scale systems by providing finer granularity of control over task execution and enabling higher level of concurrency.

However, the decomposition of workflow components into tasks presents its own set of coupling and data exchange challenges because of the enhanced granularity, added coupling, and asynchronous interaction of tasks. Data staging solutions must address these challenges to fully realize the potential of these workflows. For example, Figure 2 provides the execution and coupling of task-based data producer and data consumer applications. The execution of these applications can be represented as a directed acyclic graph (DAG), and tasks in a DAG are executed in parallel as long as no data dependency exists. In contrast with synchronous coupling (Figure 2a), the fine-grained asynchronous coupling (Figure 2b) may increase the parallelism among workflow components to improve the overall performance of the workflow. The effectiveness of staging-based approach for workflows is sensitive to the data placement across the staging cores/nodes as it can directly impact the scheduling and execution of application tasks. Appropriate data placement can leverage asynchronous coupling behaviors to maximize the potential of overlapping the execution of workflow components, so as to significantly improve the overall performance of the workflows.

In this paper, we propose an adaptive application-aware data placement approach that can efficiently support coupled application workflows requiring asynchronous coupling, such as those composed of task-based component applications. In this approach, we take advantage of asynchronous coupling patterns of the component applications to determine *which*



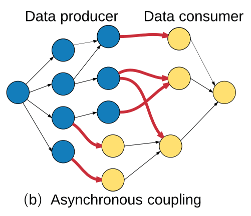


Fig. 2: Illustration of synchronous and asynchronous coupling between task-based applications.

data to place and *where* to place the data so as to optimize the parallel execution of application tasks while also reducing data movement costs, with the overall goal of improving the end-to-end performance of the workflow. Specifically, we analyze the data affinity using dataflow graphs, and use this information to place data so as to preserve locality. We also monitor and estimate the execution of application tasks on each staging core, and use this information to appropriately place data in an computation-aware manner so as to avoid load imbalance at the staging cores. Note that this data placement is performed in an *online* manner while the data is being transferred from the data producer tasks to the staging area, which are co-located with the data consumer tasks.

We have developed a prototype runtime system that implements our adaptive data placement approach on top of the DataSpaces framework and have deployed it on the Titan Cray XK7 system at the Oak Ridge Leadership Computing Facility (OLCF). We use this implementation to experimentally evaluate performance of our runtime using application workflows with two representative coupling patterns: *tightly coupled* and *loosely coupled*, and using two different applications: (a) task-based AMR (Adaptive Mesh Refinement) simulation using Charm++, (b) Topological Analysis. The effectiveness of our data placement approach, in terms of the improvement in the overall time-to-solution and the quality of data analysis, is evaluated. Results show that our approach results in up to 2.8 times speedup in the end-to-end execution time of the workflow, and an increase in the frequency of performing data analysis of 42%.

In this paper, we make the following contributions: (1) we present a data placement approach that targets coupled application workflows with asynchronous coupling patterns to adaptively place data across the staging cores with awareness of runtime workload and data locality, so as to improve the overall performance of the workflow (in terms of end-to-end execution time and data movement) and/or improve the quality of data processing; (2) we implement and deploy a runtime system based on our adaptive data placement approach on Titan, and demonstrate its effectiveness and performance using workflows with distinct coupling patterns.

The rest of this paper is organized as follows. Section 2 describes our data placement approach. Section 3 presents the design and the implementation of our runtime system. Section 4 presents the experimental evaluation. Section 5 provides related work, and Section 6 concludes the paper.

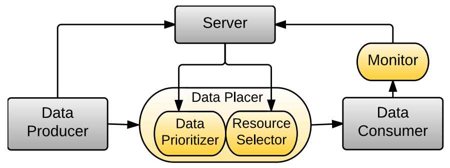


Fig. 3: Overview of the adaptive data placement approach.

II. Adaptive Data Placement

Interactions and data exchange during the execution of an application workflow using a staging-based approach proceeds as follows. Data producer tasks write data to the staging area by issuing a write request and then immediately proceed with their computation. Meanwhile, data written by these tasks is asynchronously transferred, using RDMA, to staging resources at the nodes that run the data consumer application. When this data is available in staging, a runtime scheduler flexibly schedules data consumer tasks on the cores where the required data is stored.

Thus, data placement across staging cores affects *when* and *where* the consumer application tasks are executed. Appropriate data placement should maintain data locality to avoid large amount of data movement across the network, while balancing computational load across the nodes. The data placement approach presented below considers both these aspects.

A. Overview of our approach

We design an efficient data placement approach across staging resources to support coupled application workflows with asynchronous coupling patterns. An overview of our approach is presented in Figure 3. Our approach consists of two steps. First, we determine *which* data to place based on both when the data is available, and the priority of the data relative to how much time is needed to process it. Second, we determine *where* to place the data based on application-specific data locality requirements and the runtime load at the staging cores. Typically, our approach selects staging cores that will finish data processing earlier and/or achieve lower data movement costs.

B. Data prioritization

Incoming data arrives asynchronously and is processed by our approach in a FIFO order. However, for data that arrives concurrently or within a short time window, our approach inserts each data placement request into a queue and prioritizes these requests based on their corresponding processing times, i.e., the Estimated Execution Time (EET). Typically, data with a relatively longer EET will be assigned with a higher priority, and thus will be placed onto staging cores earlier. This prevents tasks with longer execution times from becoming the bottleneck in the execution.

To efficiently estimate the execution time ($EET(d)$) of a task that consumes data d as its input at runtime, we use the observation that iterative scientific applications exhibit repetitive execution behaviors – across different time steps, tasks accessing the same data perform similar functions, and thus are likely to have a similar execution time. Therefore, we

Name	Policy Description
DATA	Place data based on data affinity to minimize data movement
TIME	Place data based on task execution time to minimize end-to-end time
HYBRID	Place data based on both task execution time and data affinity

TABLE I: Adaptive data placement policies.

monitor and compare the execution time of tasks that process the same data across consecutive time steps and use this information to anticipate their execution time for subsequent time steps. More specifically, if the monitored task execution time for data d at time steps $T-1$ and $T-2$ are $Time_d^{T-1}$ and $Time_d^{T-2}$, respectively, we estimate the execution time for data d at time step T as:

$$EET(d)^T = Time_d^{T-1} + \sigma_d; \sigma_d = Time_d^{T-1} - Time_d^{T-2}$$

This estimation mechanism works for different kinds of applications. For applications where the task runtime is proportional to the size of the input data, e.g., visualization tasks like Iso-surface Extraction and Volume Rendering, our approach captures the historical information (i.e., task execution times at the initial time steps) and uses it to estimate future execution times. For other applications where task runtimes may vary depending on the data values, like in feature tracking (i.e., topological analysis), data that contains more features may require longer execution times. Our approach leverages application-level temporal locality along with the observation that the simulation evolves predictably over time, which implies that if a feature appears in a certain data domain at the current time step, then it will likely appear in the same or nearby domains at the next time step. The evolution of features provides us with hints about the corresponding task execution times. In general, this approach is simple yet effective and introduces low overheads, as our evaluation, presented in Section 4, demonstrates.

C. Resource selection

We determine *where* to place data based on computational load and data affinity information. We refer to the *staging cores* as the *resource* in the following section.

1) Estimating computational load: To acquire the computational load at resource r as a result of placing data d, we compute the Estimated Completion Time ($ECT(d,r)$). It is calculated based on Data Available Time ($DAT(d)$), Estimated resource Available Time ($EAT(r)$) and the Estimated Execution Time ($EET(d)$) acquired above, as follows:

$$ECT(d,r) = EET(d) + max(DAT(d), EAT(r))$$

After placing data d on resource r, the *EAT(r)* of the resource will be updated accordingly.

2) Determining data affinity: Data placement determines the required data movement among tasks during the execution. For example, Figure 4 shows the dataflow graph for Topological Analysis [10] using binary reduction of a merge tree computation. Each vertex represents a task, while each edge indicates the size of the data transferred between the pair of tasks. The leaf tasks at level 1 (i.e., T_0 – T_3) read data from staging (i.e., D_1 – D_4), and communicate with tasks at level 2. Assuming that data staging is composed of two nodes, if D_3 and D_1 are placed on cores located on different nodes, T_0 and T_1 will be executed on different nodes accordingly.

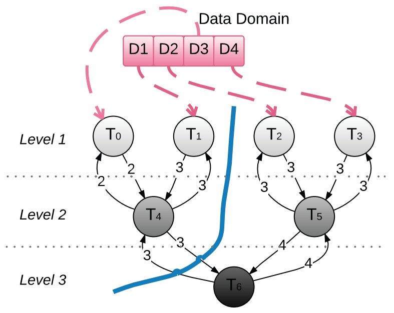


Fig. 4: An illustration of our approach for partitioning a dataflow graph. There are four data objects in the data domain, which are read by four Topological Analysis tasks, (T_0 - T_3) respectively. After partitioning, the task graph has been split into two components by the solid blue line. As a consequence, our approach identifies the pairs D_1 and D_3, and D_2 and D_4 as having high data affinity.

Consequently, no matter where T_4 is executed, the output of T_0 or T_1 has to be moved across network. Alternately, if D_3 and D_1 are placed on the same node, the data movement among T_0, T_1 and T_4 can be localized.

In most task-based runtimes, task/dataflow graphs that represent the data dependency/movement among tasks can be obtained using provided tools, such as Legion *Prof* and Charm++ *LiveViz*. Our approach leverages this information to achieve locality-aware data placement. For example, we co-locate tasks that communicate frequently with large amounts of data movement in order to localize the communication and reduce data movement costs across the network. To do this, we use METIS [9] to partition the dataflow graph into N components with the objective of minimizing data movement among components while keeping the data required by tasks in each component balanced, where N is the number of staging nodes. After partitioning, the data required by tasks in the same component are considered to have higher affinity, and are preferred to be placed together. For example, in Figure 4, the graph has been split into two components, which are mapped to two staging nodes. Based on the partitioning, it is better for D_1 and D_3, D_2, and D_4 to be placed on cores within a single node. Note that since communication patterns are always predefined and stay the same for iterative scientific applications, this operation needs to be performed only once to determine the data affinity.

D. Placement policies

Using the information described above, our approach supports flexible data placement policies to meet different data placement objectives. These policies are summarized in Table I. Users can specify their preferences as an input to select either *TIME* or *DATA*, which aim to reduce end-to-end execution time or data movement respectively. In the case of *TIME*, the resource with minimum Estimated Completion Time (*ECT*) is selected. Alternately, in the case of *DATA*, the resource with maximum data affinity is selected. If the policy is not specified by users, a *HYBRID* policy will be used by default. This policy identifies resources where *ECT* is within

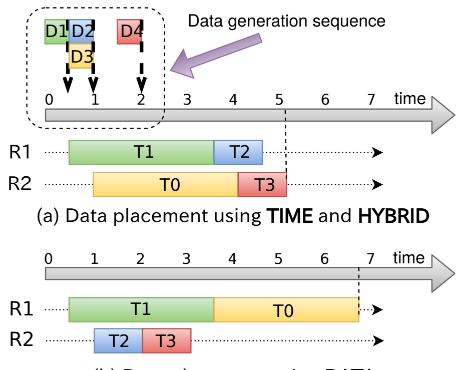


Fig. 5: An example of placement for four data ($D_1 - D_4$) onto two staging cores (R_1 and R_2), and the corresponding timing of task executions on these staging cores.

a certain range (i.e., smaller than 1.2 times of the minimum *ECT*), and then selects the one with maximum data affinity. This policy typically results in a trade-off between execution time and data movement.

To illustrate this approach, in Figure 5, four data objects arrive at timestamps 0.5, 1, 1, and 2, and tasks that read these data are shown in Figure 4. The data needs to be placed onto two resources (i.e., staging cores), and the Estimated Execution Times (*EET*) for the tasks are 3, 1, 3, and 1 for D_1 – D_4. Based on data available time and priorities, our approach places D_1 first, and then D_3 because its priority (*EET*) is larger than that of D_2. For resource selection, using the *TIME* policy, a resource with minimum *ECT* is selected (as shown in Figure 5(a)). As a result, the total execution time is 5. The *HYBRID* policy results in the same data placement as the *TIME* policy. However, using the *DATA* policy, D_3 is placed on the same core with D_1 to reduce data movement. Thus both T_1 and T_0 are executed on R_1, which is obviously overloaded compared to R_2. In this case, the total execution time is 6.5.

The time complexity of our online adaptive data placement approach is determined by the number of data requests, M, and the number of staging resources, N. We use a max-/min-heap to maintain unplaced data requests based on priority and staging resources based on *ECT* and/or data affinity. The construction of these heaps takes O(M) and O(N) time for data requests and staging resources, respectively. After initialization, we pick available data in each step and select a core to place it, which can be done in O(1). Our approach updates the corresponding heaps when the process is idle to minimize the impact on performance of runtime data placement.

III. Implementation of a Runtime System for Adaptive Data Placement

We have implemented a runtime system that incorporates our adaptive data placement approach into the DataSpaces data staging framework [6]. The schematic overview of the overall architecture of the runtime is presented in Figure 6. It leverages the *Data Communication Layer*, *Data Lookup Service*, and *Data Storage* from DataSpaces, reusing its data

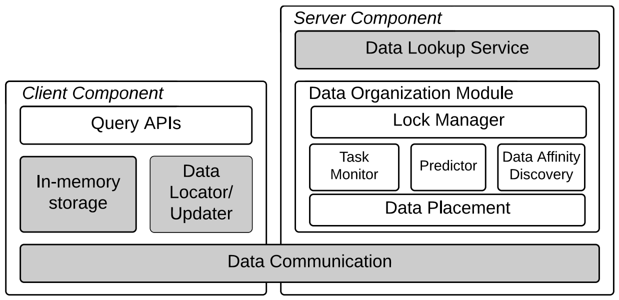


Fig. 6: Architecture of the runtime system for adaptive data placement. The shadowed area represents components of DataSpaces that are reused by the runtime.

transport, data locator, and updater capabilities. Following the DataSpaces architecture, our system consists of a client-side subsystem that is co-located with the applications in a given workflow, and a server-side subsystem that manages metadata. In this section, we describe the implementation details of the *Data Organization Module*.

A. Task Monitor and Predictor

The *Task Monitor* and *Predictor* components are responsible for monitoring and estimating the running time of tasks. Specifically, the *Task Monitor* keeps track of data read requests from the data consumer and extracts the data information (i.e., bounding box that describes the data) in the request. It logs the timing of data read requests from the same process and uses the time interval between two consecutive requests as the measurement of task execution time. The *Task Monitor* collects this information after each time step. The *Predictor* then uses this captured historical information to estimate the running time of tasks in the subsequent time steps, as described in Section 2.

B. Data Affinity Discovery

The *Data Affinity Discovery* component of our framework is responsible for identifying data dependency and movement among tasks in the *data consumer* application. It uses tools provided by a task-based runtime to obtain dataflow graph. A server selected as the master server is responsible for gathering this graph, utilizing METIS to partition the graph, and generating data affinity. Such affinity information can be used in the *Data Placement* component to reduce data movement. Since task/data dependency information does not change in iterative applications, this discovery process only needs to be performed once, after the 1^{st} time step of data consumer, and in parallel with the execution of data producer application. Therefore, it has little impact on the overall workflow performance.

C. Data Placement

The *Data Placement* component is responsible for making the data placement decision based on the information provided by the *Data Affinity Discovery*, *Task Monitor* and *Predictor* components. Specifically, each server receives the data write requests from its mapped clients and redirects them to the master server to determine placement. Based on the data

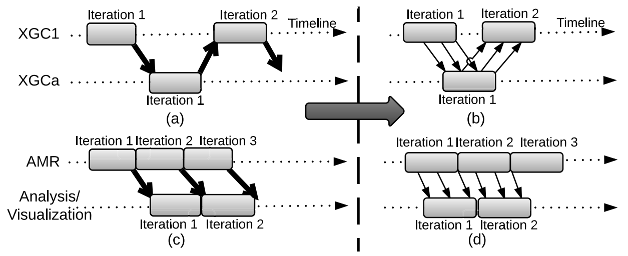


Fig. 7: Illustration of interactions and data exchanges between coupled application for XGC1-XGCa and AMR-Analysis/Visualization workflows with synchronous coupling(a,c) and asynchronous coupling(b,d).

placement approach described in Section 2, the master server selects a resource and forwards the data write request to the server that has the corresponding metadata information to process the request. After data has been put onto the staging cores, the data location is then updated and stored in a distributed hash table (DHT) constructed across all the servers. This DHT is part of the DataSpaces framework, which is detailed in our previous work [6].

D. Lock Manager

The *Lock Manager* is a routine to obtain/release exclusive rights to access the shared space and ensure data concurrency. In order to provide fine-grained and asynchronous data access, each data stored in the data storage is associated with a lock. The lock information of data is partitioned across all the servers using a similar mechanism as DHT.

For each data write/read, the task has to obtain locks for required data by sending a request to its mapped server. *Lock Manager* hashes the spatial information associated with a data write/read request and determines *which* server this data is assigned to. This server is responsible for maintaining the latest lock status (i.e., write/read lock) for all of its assigned data. Similar to other locking mechanisms, *Lock Manager* allows concurrent read and exclusive write operations. For requests that can get the lock immediately, *Data Lookup* will be invoked to locate the data and update the lock status. Otherwise, *Lock Manager* buffers the request in a queue and responds to these requests in a FIFO manner.

IV. Experimental Evaluation

In this section, we present an experimental evaluation of our application-aware adaptive data placement approach. Our experiments were performed on the Titan Cray XK7 supercomputer at ORNL. Titan has 18,688 compute nodes connected using the Gemini interconnect network; each node has a single 16-core AMD 6200 series Opteron processor and 32GB of memory.

We performed experiments to evaluate the effectiveness of our adaptive data placement approach in supporting coupled scientific workflows using representative coupling patterns: *tightly coupled* and *loosely coupled*. Figure 7a illustrates the interaction and sequential execution of two *tightly coupled* applications, using the coupled plasma fusion simulation XGC1-

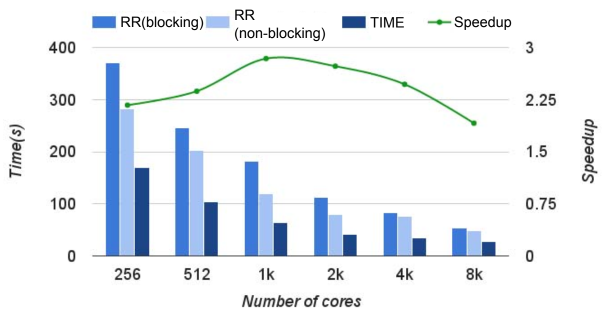


Fig. 8: A comparison of cumulative end-to-end execution time over 100 time steps using different data placement approaches. The number of cores used for AMR increased from 256 to 8192. The speedup of *TIME* over *RR(blocking)* is also plotted using the right axis.

XGCa as an example. Figure 7c illustrates the concurrent execution of a *loosely coupled* workflow using an AMR code in conjection with an analysis/visualization application as an example. In our experiments, we used two workflow scenarios with each workflow consisting of two applications coupling asynchronously. In the first test workflow, applications are tightly coupled and executed in sequential order (Figure 7b). In the second test workflow, applications are loosely coupled and run concurrently (Figure 7d). We used Adaptive Mesh Refinement (AMR), Topological Analysis, and Iso-surface Extraction as the component applications in the workflows.

A. Performance evaluation for tightly coupled workflows

To the best of our knowledge, the applications that are part of the targeted tightly coupled workflows do not yet have task-based implementations. Thus we used two task-based AMR [1] codes implemented in Charm++ to emulate the sequential execution of two applications in the tightly coupled workflow. Note that although our AMR codes are implemented using Charm++, our approach is not limited to any single task-based runtime. Furthermore, workflow components can be implemented using differing runtimes.

Charm++ implemented AMR [1] code is a finite-difference simulation of advection. It removes scalability bottlenecks in the use of collective communication in existing AMR by reformulating the communication asynchronously. Thus tasks that work on data blocks execute asynchronously, communicating only with neighboring blocks when required.

We refer to the two AMR applications composed as part of the workflow as *APP1* and *APP2*. The execution of this workflow consists of multiple coupling cycles with two-way data exchange. In each coupling cycle, the workflow first executes one time step of *APP1*, the output data are then read by *APP2* as input. Similarly, *APP1* takes the output from *APP2* to start the execution of a new time step. Both *APP1* and *APP2* read/write data from/to staging to exchange data.

Since the data exchanged between the AMR tasks is small in size, our adaptive data placement focused on reducing execution time (i.e., using the *TIME* policy) in this experiment.

Number of cores	514	1028	2056	4112	8224	16448
No. of APP1 cores	256	512	1024	2048	4096	8192
No. of staging cores	2	4	8	16	32	64
No. of APP2 cores	256	512	1024	2048	4096	8192
Domain size	1024 × 1024 × 1024					

TABLE II: This table presents the core-allocations and data

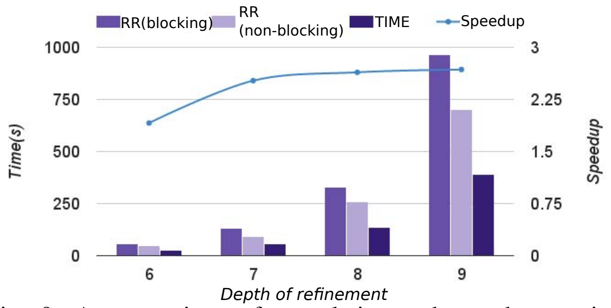


Fig. 9: A comparison of cumulative end-to-end execution time with various refinement depth ranges of AMR using different data placement approaches. The speedup of *TIME* over *RR(blocking)* is also shown on the right axis.

For comparison purposes, we compared the performance of *TIME* with static Round-Robin placement, which is commonly used for organizing multidimensional data generated by scientific applications across data staging nodes or parallel storage. We used Round-Robin in two different modes. The first is a blocking mode (*RR(blocking)*), in which data producer tasks perform a synchronous write at the end of each time step, while data consumer tasks perform synchronous reads at the beginning of their execution. Such write/read behavior is quite common among applications implemented in the traditional SPMD model. The other is a non-blocking mode (*RR(non-blocking)*), in which both data producer tasks and data consumer tasks perform asynchronous write/read. The configuration for these experiments is summarized in Table II. Detailed experimental results and comparisons are presented below. Note that these results are averages over 20 runs.

In this set of experiments, end-to-end execution time was the key metric used to evaluate the performance using different data placement approaches. The cumulative end-to-end execution time over 100 time steps is shown in Figure 8. Compared with *RR(blocking)*, both *RR(non-blocking)* and our adaptive data placement (*TIME*) achieve better performance due to concurrent execution of coupled applications. Particularly, our approach shows significant benefits in terms of the time-to-solution compared with static *RR(non-blocking)* approach. The improvement is mainly a result of the runtime workload-aware mechanism that balances the workload across staging cores. It also shows the effectiveness of our prediction of task execution time. Indeed, based on the characteristic of AMR, the task execution time is proportional to the input data size. This attribute can be efficiently captured by our framework during the initial time steps to achieve effective data placement.

In addition, we plot the speedup of *TIME* over *RR(blocking)* – the default approach used in DataSpaces. We can observe that in the case of 1k cores, our approach achieves up to 2.83 times speedup by improving the parallelism in workflow execution. However, for smaller core counts (i.e., 256), the

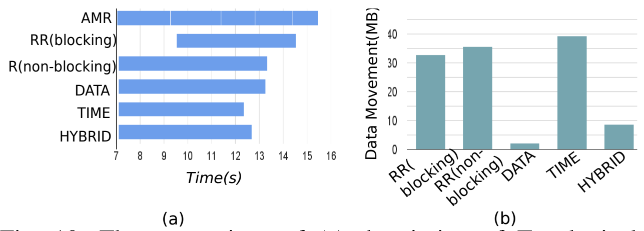


Fig. 10: The comparison of (a) the timing of Topological Analysis in one time step with respect to the execution of AMR, (b) the size of data movement during the execution of Topological Analysis in one time step, using different data placement approaches.

improvement is limited by the available staging cores, while at larger core counts (i.e., 8k), the application runs approximately one task per core, which gives us little space for improvement. Thus, the corresponding speedup is not as significant but is still larger than 1.9 times in average.

To further evaluate the performance, we used 8k cores for each AMR, and increased the refinement depth, specifically the maximum depth, which increases the number of tasks. Figure 9 plots the cumulative end-to-end execution time over 100 time steps for different depth ranges. We can observe that for a given minimal depth, increasing the maximum depth actually increases the improvement in our approach. Compared to *RR(blocking)*, our approach attains a speedup of 1.9 when depth ranges from 4 to 6, and the speedup increases to 2.68 when the depth ranges from 4 to 9. The results shows that the potential benefits of parallelism gained by using our data placement approaches grows as the number of tasks increases.

System overheads: System overheads are mainly due to runtime data placement across staging cores. We have identified three primary sources of overheads: (1) time to acquire task execution time, (2) time to collect task runtime information, and (3) time to determine data placement. We measured the overheads of data placement, which is smaller than 0.001 second in average for each placement request, thus can be considered negligible when compared to simulation time. In addition, most of these overheads come from the communication and computation among metadata servers, which can be performed in parallel with the applications' computation. Therefore, the introduced overhead would not have too much impact on the end-to-end execution time of the workflow, as shown in the experiments.

B. Performance evaluation for loosely coupled workflow

In this experiment, we used AMR in conjection with Topological Analysis workflow. Different from the previous experiments, the two coupled applications are running concurrently. AMR writes data while Topological Analysis reads the data and performs online data processing without impacting the execution of AMR.

In the loosely coupled workflow, using different data placement approaches may not have much impact on the end-to-end time of workflows because simulation time is always the dominant factor. However, it may impact the effectiveness of an execution from a science perspective. For exmaple, the frequency at which data analysis is performed impacts the ability to capture intermittent phenomena. In this set of experiments, we measured how often can analysis be performed, without significant impact on the end-to-end execution time. The configuration for these experiments was as follows: the number of cores allocated for AMR and Topological Analysis was 2048 and 256 respectively, with 16 cores used for the metadata servers. The data domain used by AMR was $1024 \times 1024 \times 1024$.

The Topological Analysis code was developed by Lawrence Livermore National Laboratory using a k-way hierarchical reduction algorithm [10]. This algorithm involves four kinds of tasks: *local computation*, *join*, *correction*, and *output*. *Local computation* tasks read data from simulation and perform purely independent computation as soon as data is available, i.e., asynchronous read patterns. *Join* and *correction* require inter-task data movement within a group of tasks. In our study, we set $k = 8$, which is commonly used to perform a more spatially coherent merge. The dataflow graph is similar to the binary reduction dataflow with $k = 2$ shown in Figure 4 in Section 2.

In this experiment, we took both execution time and data movement into consideration and measured the performance of our adaptive data placement approach using the different policies, i.e., *TIME*, *DATA* and *HYBRID*. A detailed description of these policies is in Section 2. We also compared them with the Round-Robin approaches, i.e., *RR(blocking)* and *RR(non-blocking)*.

Figure 10a shows the timing of Topological Analysis at one time step using different data placement approaches. The execution time of AMR is also shown in the figure as the baseline for comparison, which starts from 11^{th} time step and includes 5 time steps in total. As expected, using *TIME*, Topological Analysis starts and also finishes much earlier than using *RR(blocking)*, which enables it to analyze simulation data more frequently.

Comparing the results of adaptive approaches using different placement policies, using *TIME*, Topological Analysis finished earlier than using *DATA*, but generated a larger amount of data movement, as shown in Figure 10b. This is because better data locality may result in load imbalance across staging cores, causing a longer execution time. In comparison, using *HYBRID*, the execution time is a little longer than *TIME*, but the size of data movement is 8.7MB, compared to 39.4MB for *TIME*. These results are consistent with the objectives of the different policies. Note that although the size of the data movement is not large in one time step in this case, for communication-/data-intensive workflows running thousands of time steps, reducing data movement can significantly improve the performance of the workflow by alleviating network contention/ congestion.

After measuring the performance of Topological Analysis at one time step, we also measured how often Topological Analysis could be performed during a simulation run of 100 time steps. We varied the number of cores used for Topological analysis to 256 and 512. As shown in Figure 11, we achieved the highest possible frequency for performing Topological Analysis, i.e., at every time step, using *TIME* in the case of 512 cores. In contrast, using *RR(blocking)*, we were able to perform analysis at every time step in the first 40 time steps, and at every other time step in the last 60 time steps due to the increasing volume of simulation data, thus 70 times in total.

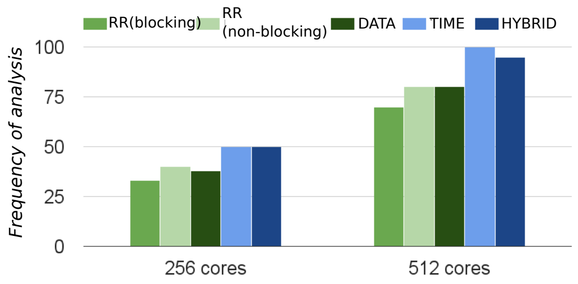


Fig. 11: A comparison of the frequency of Topological Analysis that can be performed during 100 time steps of simulation using different data placement approaches. The number of cores for Topological Analysis is 256 and 512, respectively.

V. Related Work

Task-based runtime and applications: Recently, asynchronous many task (AMT) models and runtime systems have received a lot of attention [8], [3]. There are many production applications that have already been implemented using task-based runtimes, such as S3D with Legion [8] and NAMD with Charm++ [13]. Recent researches [3] mainly focus on workflows composed of applications that are implemented in a single task-based runtime. In this manner, the built-in runtime scheduler can efficiently support the code coupling with the awareness of the coupled applications in a workflow. However, supporting a wide-range of applications across a variety of disciplines can result in a diverse set of requirements where components of the same workflow may utilize different task-based runtimes. In this case, the built-in schedulers cannot coordinate with each other conveniently and efficiently. To address this, our work aims to provide a more generic solution – using data staging – to efficiently support the code coupling for a wider range of scientific workflows.

Data management in scientific workflow management systems: Traditionally, intermediate files within a workflow are stored in shared storage systems or distributed storage systems, such as Swift [15] and Pegasus [5]. This approach can be easily used for coupling scientific application but may introduce significant I/O overheads. Also, shared storage APIs, i.e., POSIX, constrain the communication between the storage and runtime systems which prevents locality-aware task scheduling.

Some researches have proposed coupling datastore services with a workflow runtime engine, such as Falkon [11]. This approach gives up the layered design, but is still designed and optimized for a single application or for applications implemented within the same runtime. Supporting component applications using different runtimes remains challenging due to the distinct requirements imposed on external storage systems. Our data placement approach aims to bridge that gap in order to support efficient coupling for scientific workflows.

In-staging data placement: A number of data-staging frameworks, such as DataSpaces [6] and Flexpath [4], have been developed in recent years to support scientific workflows using memory within a data stagng area for runtime data coupling and exchange. Our previous work [12] optimize the data placement in staging based on network topology to reduce data access overheads. However, these works have focused on workflows with synchronous coupling of applications implemented in SPMD mode, thus can be inefficient for supporting asynchronous coupling exhibited by task-based applications, as demonstrated in the experiments.

Workflow-aware storage system: Several research efforts target workflow-based applications, which have similar fine-grained and asynchronous coupling among tasks. AMFS [16] and MemFS [14] propose a distributed in-memory runtime file system. Since AMFS only issues local reads/writes to utilize the high bandwidth of local memory, it may lead to load imbalance, which impacts performance and scalability. In contrast, MemFS scatters data among all the nodes to achieve a well-balanced workload storage for executing tasks, based on the assumption that the data movement costs across the network are negligible. However, in current HPC systems like Titan, the network contention/congestion can significantly impact the data access performance [12]. Our runtime system takes both data locality and load balancing into consideration to optimize the data placement.

VI. Conclusion

In this paper, we propose an in-staging data placement approach to improve time-to-solution for task-based scientific workflows. Our approach leverages the asynchronous coupling of applications to optimize the data placement in a staging area based on application-specific data locality requirement and runtime performance of task executions on staging cores/nodes. This paper also presents the design and implementation of a runtime system that realizes our approach. We have deployed this runtime on Titan Cray XK7 system and experimentally evaluated it with representative scientific workflows. The results demonstrate that our data placement approach can effectively improve the end-to-end execution performance, reduce overall data movement, and also increase the quality of scientific insights.

VII. Acknowledgments

The research presented in this work is supported in part by National Science Foundation (NSF) via grants numbers ACI 1339036, ACI 1310283, CNS 1305375, and DMS 1228203, by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the US Department of Energy Scientific Discovery through Advanced Computing (SciDAC) Institute of Scalable Data Management, Analysis and Visualization (SDAV) under award number DE-SC0007455, the Advanced Scientific Computing Research and Fusion Energy Sciences Partnership for Edge Physics Simulations (EPSI) under award number DE-FG02-06ER54857, and the ExaCT Combustion Co-Design Center via subcontract number 4000110839 from UT Battelle. This used resources of the Oak Ridge Leadership Computing Facility is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. The research at Rutgers was conducted as part of the Rutgers Discovery Informatics Institute (RDI2).

References

[1] Charm++ amr. url=http://charmplusplus.org/miniApps/amr.

[2] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale. Parallel programming with migratable objects: Charm++ in practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14. IEEE Press, 2014.

[3] K. Chandrasekar, B. Seshasayee, A. Gavrilovska, and K. Schwan. Task characterization-driven scheduling of multiple applications in a task-based runtime. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*, ESPM '15, New York, NY, USA, 2015. ACM.

[4] 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 *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 246–255. IEEE, 2014.

[5] 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. *Sci. Program.*, 13(3), July 2005.

[6] C. Docan, M. Parashar, and S. Klasky. Dataspaces: An interaction and coordination framework for coupled simulation workflows. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, New York, NY, USA, 2010. ACM.

[7] C. Docan, F. Zhang, T. Jin, H. Bui, Q. Sun, J. Cummings, N. Podhorszki, S. Klasky, and M. Parashar. Activespaces: Exploring dynamic code deployment for extreme scale data processing. In *Concurrency and Computation: Practice and Experience*, 2014.

[8] R. C. J. Wilke, J. Bennett. Enabling runtime/application co-design through common concurrency concepts. In *Runtime Systems for Extreme Scale Programming Models and Architectures (SC15 Workshop)*, 2015.

[9] G. Karypis and V. Kumar. Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[10] A. G. Landge, V. Pascucci, A. Gyulassy, J. C. Bennett, H. Kolla, J. Chen, and P.-T. Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, Piscataway, NJ, USA, 2014.

[11] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: A fast and light-weight task execution framework. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 43:1–43:12, New York, NY, USA, 2007. ACM.

[12] Q. Sun, T. Jin, M. Romanus, H. Bui, F. Zhang, H. Yu, H. Kolla, S. Klasky, J. Chen, and M. Parashar. Adaptive data placement for staging-based coupled scientific workflows. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. ACM.

[13] Y. Sun, G. Zheng, C. M. E. J. Bohm, T. Jones, L. V. Kalé, and J. C.Phillips. Optimizing fine-grained communication in a biomolecular simulation application on cray xk6. In *SC'12*, November 2012.

[14] A. Uta, A. Sandu, S. Costache, and T. Kielmann. Scalable in-memory computing. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, May 2015.

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

[16] Z. Zhang, D. S. Katz, T. G. Armstrong, J. M. Wozniak, and I. Foster. Parallelizing the execution of sequential scripts. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, New York, NY, USA, 2013. ACM.