Integrating External Resources with a Task-Based Programming Model

Zhihao Jia
Stanford University
zhihao@cs.stanford.edu

Sean Treichler
NVIDIA
sjt@cs.stanford.edu

Galen Shipman
Los Alamos National Laboratory
gshipman@lanl.gov

Michael Bauer
NVIDIA
mebauer@nvidia.com

Noah Watkins
UC Santa Cruz
jayhawk@soe.ucsc.edu

Carlos Maltzahn
UC Santa Cruz
carlosm@soe.ucsc.edu

Patrick McCormick
Los Alamos National Laboratory
pat@lanl.gov

Alex Aiken
Stanford University
aiken@cs.stanford.edu

***Abstract*—Accessing external resources (e.g., loading input data, checkpointing snapshots, and out-of-core processing) can have a significant impact on the performance of applications. However, no existing programming systems for high-performance computing directly manage and optimize external accesses. As a result, users must explicitly manage external accesses alongside their computation at the application level, which can result in both correctness and performance issues.**

We address this limitation by introducing Iris, a task-based programming model with semantics for external resources. Iris allows applications to describe their access requirements to external resources and the relationship of those accesses to the computation. Iris incorporates external I/O into a deferred execution model, reschedules external I/O to overlap I/O with computation, and reduces external I/O when possible. We evaluate Iris on three microbenchmarks representative of important workloads in HPC and a full combustion simulation, S3D. We demonstrate that the Iris implementation of S3D reduces the external I/O overhead by up to 20×, compared to the Legion and the Fortran implementations.

I. Introduction

Most existing programming systems for high-performance computing (e.g., Legion [1], Sequoia [2], Charm++ [3], and X10 [4]) have a *closed world* data model: all program data are created, managed, and accessed exclusively inside the runtime, and no external applications or systems can manipulate the data. Of course, programs written in these systems do routinely share data with external systems. For example, scientific simulations checkpoint intermediate results into storage for external applications to analyze. As another example, a rendering application may load data from an external application and generate dynamic visualizations. These external interactions are done outside the semantics of the programming model and therefore are treated conservatively by the compiler and/or runtime system.

To the best of our knowledge, no existing parallel systems have defined semantics for runtime management and optimization of interactions with external data. Systems we surveyed defer the responsibility to applications, which results in both correctness and performance issues:

- **Data consistency** must be explicitly controlled at the application level. Applications often copy large portions of external data into local memories for efficient processing. Current systems require applications to explicitly manage updates to these in-memory copies and their propagation to the external resource. We define a consistency model for external data, reducing application complexity and the potential for errors when accessing external data.
- **Overlapping communication and computation** is difficult if the programming system does not have knowledge of and control over external data transfers. For example, most task-based parallel systems delay the execution of tasks until all input data is in place. If external data must be accessed by the application inside of compute tasks, this may cause those tasks to block and result in poor performance. When multiple tasks contend over the same external data this issue can become quite acute with degraded performance and the potential for deadlock in some systems.
- **Performance portability** is more difficult to achieve when applications are forced to explicitly manage both synchronization and overlap of computation with access to external data. System attributes such as processor throughput, memory bandwidth, and I/O bandwidth vary significantly from one HPC system to another. As a result, performance portability is greatly reduced.

To address these challenges we first introduce *external resources*, a model for data that is not exclusively owned and accessed by a single application or runtime. Examples of external resources are file systems, databases, and key-value stores. Next we present Iris, a *deferred execution* programming model with explicit semantics for managing external resources. Iris is designed around three core concepts:

- Data model: Iris's *regions* are data collections that serve as inputs/outputs of computation. Regions are abstracted from the physical resource which may hold the data of the region. Each external resource is modeled as one or more regions.
- Coherence model: Iris provides explicit *coherence models* for accessing regions. External resources are regions

with *simultaneous coherence*, meaning that there may be multiple simultaneous readers and writers of the region, and all updates are visible to all readers/writers. The readers and writers for external resources can include processes external to Iris as well as Iris tasks.
- Consistency model: A task may *acquire* a region with simultaneous coherence to indicate when it is safe to make copies of the region's data (e.g., to make an in-memory copy of data on disk for higher performance). A *release* flushes updates to in-memory copies back to the external resource, providing release consistency.

Iris builds on Legion's [1] existing mechanisms for managing regions and therefore inherits many of the Legion optimizations which we extend to external resources. The main contributions of this paper are as follows:

- We propose Iris, a deferred execution programming model with explicit semantics for external resources.
- We present three important optimizations in Iris that maximize I/O bandwidth utilization, extract additional task parallelism and minimize data transfers.
- We evaluate Iris on three microbenchmarks and a full combustion simulation, S3D. We show that, compared to the Legion and Fortran implementations, the Iris version of S3D reduces the I/O overhead by up to 20×.

II. Programming Model

A core tenet guiding our design of Iris was to integrate well with Legion's asynchronous task-based programming model with minimal changes. This was motivated in part by our desire to provide the same semantics for internally as well as externally managed resources, eliminating the need for the application developer to reason about distinct, perhaps orthogonal, semantics for these resources. This unified model also takes direct advantage of many of the underlying optimizations employed by the Legion implementation. While our current work focuses on Legion, many of the ideas in Iris are applicable to other programming systems such as Sequoia [2], Charm++ [3], and X10 [4]. We discuss this in more detail in Section VII.

A. Legion's Regions

In Legion, a *region* names a data collection. Regions are first-class values in Legion and serve as the inputs and outputs of computation [1]. It is worth emphasizing that regions only name a data set. Regions are also mutable: computations may directly modify the contents of regions. Regions do not have to be materialized at all times, nor do they commit to any particular data layout or physical placement in a machine. A *physical instance* of a region is a copy of the region's data with a particular data layout and stored in a specific place (e.g., on disk, in DRAM, in a GPU framebuffer, etc.). There may be 0, 1, or multiple physical instances of a region in existence at any time within Legion. Allowing multiple instances of a region is necessary for many optimizations, such as allowing multiple readers to each have a local copy of the data for more efficient access.

B. Legion's Tasks

Tasks are functions that perform computation on one or more regions. Each task maintains two properties for regions on which they operate:

- *Privileges* specify how the task will access each region, which include read only, read-write, and reductions with a commutative reduction operator.
- *Coherence* specifies how other potentially concurrent tasks may access the same region: *exclusive access* (task ordering must be preserved and no other task can appear to access the region concurrently), *atomic access* (tasks may be reordered, but no other task can appear to have concurrent access), and *simultaneous coherence* (visible updates from concurrent tasks are permitted).

Tasks are issued to the Legion runtime system in sequential program order, but the underlying runtime system is free to perform optimizations that may reorder tasks and execute them in parallel provided the results are consistent with the serial order and the declared coherence modes on regions.

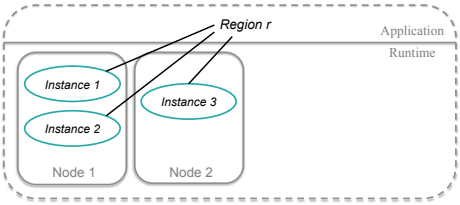
C. Legion's Deferred Execution

Deferred execution decouples the launching of tasks from when the tasks are executed. The task launch operation returns immediately, and tasks are executed asynchronously by the runtime. Asynchronous launches can be composed: the results of launching task A may be passed as arguments to the asynchronous launch of task B, even if A has not yet completed. [1] Therefore, unless the program needs the result of a task launch to make a control-flow decision, all tasks are launched without any delay and tasks are never required to block on any Legion operation. Deferred execution allows Legion to hide long-latency operations by launching tasks well in advance of when they actually run, which allows Legion to discover other independent work that can be executed at the same time.

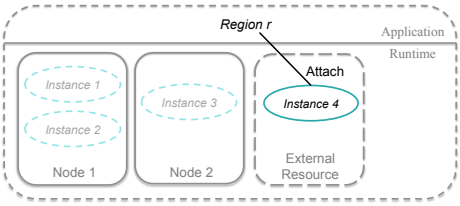
D. Iris's External Resources

In Iris, we model data and memory that is not exclusively owned and accessed by Iris as an *external resource*. A simple example of an external resource is a formatted file (e.g., an HDF5 file on disk) shared between an Iris application that checkpoints intermediate results into it and an external application that analyzes the results. Other examples of external resources include objects in key-value storage systems, databases in persistent storage, and opaque data produced by external applications. There are three problems we must address:

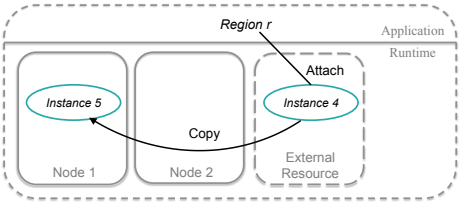
[1] As an aside, this design may sound like *futures*, but not all versions of futures allow task calls to be composed—i.e., in some models passing a future as a function argument blocks on the result of the future.



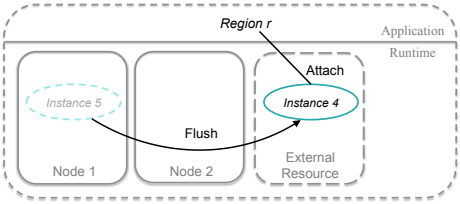
(a) Step 0: Region r is previously mapped to 3 physical instances on different nodes.



(b) Step 1: *Attach* invalidates existing physical instances of region r, and maps it to an external resource.



(c) Step 2: *Acquire* removes the copy restriction on region r and allows tasks to create local copies.



(d) Step 3: *Release* invalidates local copies and flushes updates back to the external instance.

Figure 1: Using an external resource in Iris.

- New primitives are required to associate an external resource with a region and also to remove such an association. We introduce two new operations, *attach* and *detach*, in Section II-E.
- Iris tasks and external tasks may access the external resource concurrently necessitating some form of explicit synchronization. We use existing Legion operations for regions with simultaneous coherence, *acquire* and *release*, for this purpose. While these operations predate Iris, their semantics have not been previously presented in the literature; we provide an overview in Section II-F.
- Supporting arbitrary external resources requires a mechanism for supporting arbitrary data formats/layouts. Our solution is presented in Section II-G.

E. Attach and Detach

Attach integrates external resources into Iris by associating an external resource with a region. For example:

```
attach(r, file)
```

associates the region r with an external file. There are different versions of `attach` for each kind of external resource (see Section V-A).

In Legion, a region is always associated with some data set (though this may be empty) and is always associated with zero or more physical instances. Thus, when attached to an external resource, the region must first be disassociated from its existing data. Attach *invalidates* all existing physical instances associated with the region and makes the external resource the only valid physical instance of the region.

Figures 1a and 1b illustrate an example of attach in Iris. Before the attach operation, region r is mapped to three physical instances. The attach operation associates r with data in an external resource, as shown in Figure 1b. Attach invalidates all three existing instances and maps region r to the newly created external physical instance (i.e., instance 4). All subsequent tasks that use region r will now refer to instance 4. It is worth noting that no actual external I/O occurs as a result of the attach operation.

The *detach* operation disassociates a region from its external resource. The Iris runtime defers the detach until all tasks using the attached region have completed.

F. Acquire and Release

As discussed in Section I, the regions attached to external resources have simultaneous coherence. Part of the semantics of simultaneous coherence is that tasks can see the updates of other tasks concurrently accessing the region, which implies that all the tasks use a single common physical instance. This design is appropriate in many situations but prevents optimizations that rely on cached local copies. For example, if an external resource resides on disk then by default Iris tasks must perform disk I/O every time they access the external resource. This can result in poor performance for tasks that make frequent changes to regions.

In Legion, a task can create local copies (i.e., additional instances in faster memories) of a region r with simultaneous coherence by first issuing an *acquire* operation on r. The acquire indicates that no external task will access the region during the lifetime of the acquire so it is safe to allow the runtime to make copies of the data. In Iris, an acquire operation on a region attached to an external resource allows tasks to operate on local copies of the external data.

When all tasks that use local copies have finished, a *release* operation invalidates all local copies by flushing any updates on these copies back to the original instance. The external resource is thereby brought back to a consistent state with the task updates to these temporary copies.

The semantics for acquire and release are similar to release consistency [5]: all writes between acquire and release are visible to other tasks upon the release operation.

Figure 1c and 1d illustrate acquire and release. In Figure 1b, region r is attached to an external resource (i.e.,

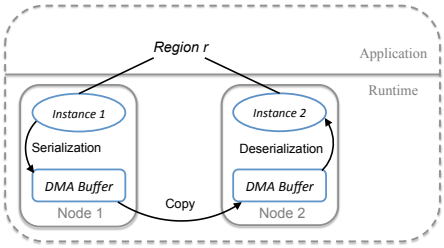


Figure 2: Transferring opaque data structures in Iris.

instance 4). All subsequent tasks that use region r before an acquire operation access the external resource directly. In Figure 1c, the application launches an acquire operation which removes the copy restriction. Tasks that need r as input can then use local copies (i.e., instance 5). When the application has finished launching all tasks that need local copies of region r, it launches a release operation that flushes any local changes back to the external resource, and invalidates all local copies[2], as shown in Figure 1d.

G. Opaque Data Structures

Iris supports attaching external data that is *opaque* to Iris, meaning its data layout is not understood by the runtime. In addition to providing the generality to support many external resources, there are specific use cases where opaque external resources are appropriate. For example, applications with security restrictions may need to hide the actual data from Iris and only expose a narrow interface. Another example is when the "data" is actually another application that only produces the data on demand. Finally, a common use case is a complex data structure whose format is application dependent and which may vary depending on input parameters, making encoding this format directly within Iris undesirable.

To allow tasks to access opaque external resources, Iris provides a *serdes* (serialization-deserialization) interface for applications to describe how to serialize regions attached to opaque external resources into runtime managed buffers and how to deserialize these buffers when creating a new physical instance of the region in a destination memory.

Figure 2 illustrates how Iris uses the serialization/deserialization mechanism to copy instances associated with opaque data structures. Instance 1 is initially attached to a collection of binary trees owned by an external application. The elements of the region r are just references to the roots of the trees, and tasks call functions in a binary tree library to do any required tree manipulations.

Note that to make a copy of the region, it is not sufficient to simply copy the pointer contents—the binary trees themselves, which are external to the region, will need to be copied if the region is copied to a different address space (i.e., a different node). In Figure 2, when copying instance 1 to a remote node, Iris first serializes instance 1 to a *DMA buffer* on the local node. The DMA buffer contains all necessary information to reconstruct the binary trees. The Iris runtime then performs a bit-wise copy to transfer the DMA buffer to the destination node. Finally, the Iris runtime reconstructs a new instance (i.e., instance 2) on the destination node by deserializing the DMA buffer.

An additional advantage of using a serialization/deserialization interface is that it allows highly compressible data to be transferred with minimum network bandwidth by first serializing the opaque structure into a compressed form and then deserializing it at the destination. Furthermore, the serialization/deserialization interface provides an additional mechanism for layout optimizations when moving data between internal and external resources in Legion.

[2]Iris runtime defers the actual invalidation of local copies until the local copies have expired. By doing this, the Iris runtime eliminates data copies in the case where the application re-acquires the same region.

```
//all databases are pre-attached to regions
while (!input.empty()) {
    struct TaskInfo t = input.next_task();
    Region r = t.region;
    acquire(r);
    if (t.type == QUERY) {
        query(t, r);
    } else {
        update(t, r);
    }
    release(r);
}
//detach all regions after all tasks are done
```

Figure 3: Database application kernel task in Iris

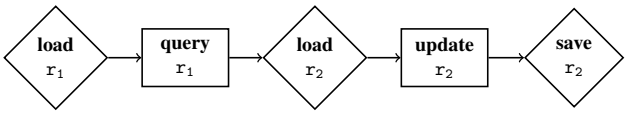
III. An Example

To illustrate the design of Iris, we describe an application that interacts with multiple external relational databases stored on disk. The application accepts a stream of queries (read-only operations) and updates (read-write operations) to particular databases. To implement the application in Iris, each database is modeled as a region and queries/updates are modeled as tasks that operate on these regions.

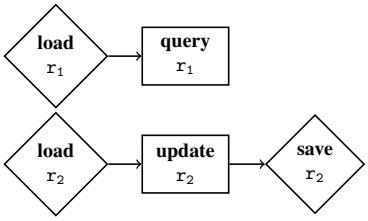
In Figure 3 we present pseudo-code for the core application loop in Iris. Here, `input` is a queue of input tasks to be processed. `TaskInfo` is a structure that includes all necessary information for performing a task. The loop iteratively pulls new tasks from the queue and launches the corresponding operations based on task information. Every task is either a `query`, which needs `read-only` privileges on the input database, or an `update`, which requires `read-write` privileges.

For performance reasons, the query/update tasks copy the database contents into local DRAM before executing their application logic (not shown) and flush any changes back to the external databases during `update` operations. To enable local copies the main loop issues a pair of `acquire` and `release` operations before and after launching the task.

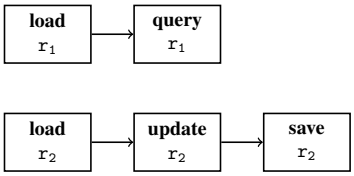
Depending on the specific order of queries/updates to databases, a number of optimizations are possible. We illustrate two optimization scenarios. Figure 4 shows an execution example where the application performs a query and an update on different databases (i.e., r_1 and r_2). In the figure, `load` and `save` indicate where actual external data is loaded from and saved to the disk. Figure 4a shows



(a) Execution graph in serial programming execution.



(b) Execution graph in deferred execution *with* external resource integration.



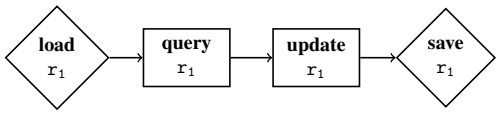
(c) Execution graph in deferred execution *without* external resource integration.

Figure 4: An execution where the application performs a query and an update on different database. Rectangles refer to Iris tasks, diamonds represent internal Iris operations, and solid arrows indicate data dependencies.

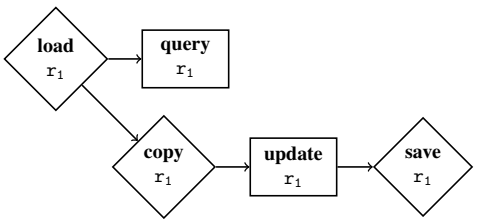
the serial execution order of operations. Because operations on region r_1 and r_2 do not interfere (because the data is from different databases), Iris can perform the operations in parallel, as shown in Figure 4b.

We argue that our approach to integrating external resources is especially beneficial when coupled with deferred execution. Without external resource semantics in Iris, applications must explicitly perform external I/O within the Iris tasks. Figure 4c shows an example in which the application defines additional Iris tasks (illustrated as rectangles "load" and "save") for loading and saving data in external databases. Performing explicit external I/O inside Iris tasks can degrade the application's overall performance because: (1) the processors must block on external I/O, especially in the case when multiple processors are competing for external bandwidth; and (2) mixing computation with I/O makes it difficult to overlap the I/O with computation. Iris addresses both issues since all external I/O becomes internal data transfers which do not block compute tasks and can be scheduled to overlap with computation.

In addition to the benefits of deferred execution, Iris can extract additional parallelism even in the case where the task execution order is restricted by data dependencies. Figure 5 illustrates another execution where the application performs a query and then an update on the same database. Using only a single instance of the data, we cannot achieve parallelism as the ordering of the two tasks cannot be changed. Reordering the two operations could result in the `query(`r_1`)` task observing changes made by the `update(`r_1`)` task, which could affect the result of the `query(`r_1`)`.



(a) Execution graph *without* the write-after-read (WAR) optimization.



(b) Execution graph *with* WAR optimization. Iris launches an additional copy to create another local instance.

Figure 5: An execution where the application performs a query and then an update on the same database.

However, if the `query` and `update` tasks operate on different local copies they can be executed in parallel, and doing so won't affect the program's final state (as shown in Figure 5b). It is worth noting that even if Iris launches an additional copy operation to create a second local copy for parallel execution, this doesn't necessarily indicate Iris incurs the cost of additional external I/O. The `load` creates a local copy inside the Iris runtime, and Iris could use this copy as the source data for the second in the likely case that that is cheaper than performing external I/O.

IV. Optimizations

In this section we discuss three important optimizations for accessing external resources in Iris. Two of these, *deferred execution* and *write-after-read* were illustrated in Section III.

A. Deferred Execution

Iris decouples external I/O from computation and maps them to separate operations. For example, in Figure 4b, a single database update operation is decomposed into `load`, `update`, and `save` operations in Iris. As a result, external I/O and computation are executed by different threads dedicated to certain types of Iris operations. In our implementation, Iris has dedicated DMA threads for handling data movement (including both data copies within the Iris runtime and accessing external resources) and compute threads for executing application tasks.

Tasks are not expected to perform external I/O in Iris. Instead, tasks attach external resources to regions and let the runtime optimize external I/O. This separation allows Iris to perform several scheduling optimizations, including:

- **Better CPU utilization.** Compute tasks that need to access external resources do not block on external I/O and therefore do not tie up CPU resources while external I/O is being performed.
- **Global Management of Resource Usage.** All accesses to an external resource are handled and performed

by the Iris runtime allowing Iris to reschedule I/O to improve overall application performance.

Iris knows about *all* I/O operations for every media that holds external data. With this knowledge Iris can reschedule data transfers to maximize bandwidth utilization and minimize I/O contention. This I/O rescheduling could not be performed at the application level as the application is unaware of I/O performed by the runtime system itself. For the database example in Section III, the runtime system may also use the same disk for checkpointing, resulting in disk I/O invisible to the application. If the application issues database save/load operations assuming it has the entire disk bandwidth, both database accesses and checkpoints are degraded, leading to poor overall performance.

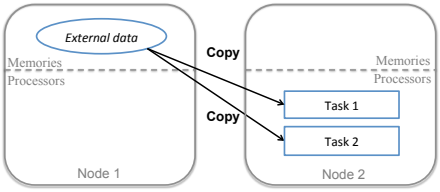
Even in the absence of conflicting I/O operations within the runtime, managing concurrent application-level access to external resources directly within the application is difficult. Applications often defer the complexity to middleware libraries [6] [7], which can intercept I/O operations made by the application as they are invoked. While beneficial, this method is unable to reorder I/O operations "in time" based on application-level data dependencies, as can be done in Iris, as these models do not defer operations. Deferring execution of tasks as well as I/O operations managed directly in the runtime allows Iris to fuse concurrent I/O operations while preserving data dependencies.

B. Write-after-read

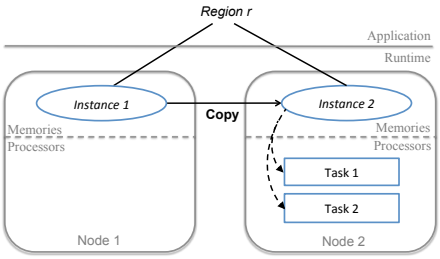
In the case where all write tasks launched between an acquire/release pair have been completed (i.e., no further updates will be made to the local copies of the external resource), Iris can allow unfinished read-only tasks to use a local copy and flush dirty data produced by write tasks back to external resource by performing a release. This early release is safe because: (1) all write tasks have been completed, which guarantees that the update to the external resource is the same as it would be after all tasks are completed, (2) all unfinished read-only tasks (if any) will still observe the correct data using the local copies, even though data in the external resource has been modified, and (3) future acquire operations will create new local copies of the data from the external resource.

C. Reducing Data Transfers

When launching tasks that access external resources, Iris can choose any up-to-date instance based on locality and availability, efficiently eliminating redundant or avoidable data transfers. Without external resource semantics, applications would have to manage all I/O in tasks, where the straightforward approach is to use blocking I/O operations and incur the full overhead of reading and writing to external resources. An alternative is to manage data dependencies between tasks and their external resources directly at the application level at the cost of great complexity.



(a) Without external resource semantics, all tasks perform individual in-task data transfers.



(b) Iris manages external resources and local copies, allowing tasks to use local copies when available.

Figure 6: An example that illustrates reducing data transfers.

Figure 6 shows an example in which two tasks access external data on a remote node. Without external resource semantics, the application cannot represent the external data inside the programming model. Therefore, the only option is that all tasks perform individual in-task data transfers (shown as two arrows in Figure 6a). To eliminate redundant data transfers, application developers can implement a library on top of external resources to track all up-to-date copies and direct tasks to use local copies if available. However, one limitation is that it is only able to reason about copies created in the application and is blind to copies made by the runtime.

Iris provides a straightforward way to reduce data transfers. Figure 6b shows an execution for the above example in Iris. The external data is labeled as an instance (i.e., instance 1) of region r after the attach operation. Iris tracks all data dependencies and is aware that the two tasks on node 2 need region r as an input. Consequently, the Iris runtime creates a local copy (i.e., instance 2) of region r on node 2 and direct the two tasks to use the local copy. The application may launch future tasks that manipulate region r on node 2. As long as the local copy (i.e., instance 2) stays up-to-date, all future tasks on node 2 can directly exploit the local copy without additional data transfers.

V. Implementation

We have implemented Iris in Legion [1], a high-performance parallel runtime for distributed architectures.

A. API Extension

To support external resource semantics, we have extended the Legion API with attach and detach operations. We currently support the following external resources: normal

```
PhysicalInstace attach(
    char *filename,
    Region region,
    const std::map<FieldID, char*> &fieldmap,
    HDF5AccessMode mode);
```

Figure 7: Code snippet for attaching a HDF5 file.

files, HDF5 (hierarchical data format [8]) files, relational databases, and other opaque data structures.

Figure 7 shows the C++ code snippet for attaching an HDF5 file. This `attach` method attaches an external HDF5 file named `filename` to an existing region identified by `region` and returns the physical instance that represents the external data. A feature of Legion regions that we have not yet explained is that the elements of regions can themselves have structure [9]. In particular, each region has a set of *fields*, and elements of the region can be thought of as objects or structs with those fields. The `fieldmap` expresses mapping from HDF5 datasets to different fields.

To support integration with opaque data structures we have added custom serialization/deserialization operations to the Legion API for copying opaque data structures. To specify serialization/deserialization for a data type in Iris, users must provide the following four methods:

- `size`, which computes the number of bytes needed for serializing a single element.
- `serialize`, which serializes the element into a buffer provided by the Legion runtime.
- `deserialize`, which deserializes a buffer argument into an opaque data structure.
- `destroy`, which destroys all data associated with the element argument.

B. Runtime Extensions

In addition to modifying the Legion API, we also extended the Legion runtime to support attach and detach.

To support copies of opaque data structures we have extended the Legion data transfer subsystem by adding serialization and deserialization components. The original Legion data transfer mechanism supported only bit-wise copies between memories. Moving a physical instance registered with custom serialization/deserialization operations requires first serializing the data into a DMA buffer. Next, the DMA buffer is transferred to the destination node. Finally, the runtime deserializes the DMA buffer to reconstruct a new physical instance in the destination memory.

VI. Evaluation

In this section we evaluate the performance of Iris and the effectiveness of the three optimizations described in Section IV by considering the effect of each optimization individually as well as together.

We first consider three microbenchmarks implemented in Iris: matrix multiplication (Section VI-A), a database query/update simulator similar in spirit to the application presented in Section III (Section VI-B), and a distributed scene rendering application (Section VI-C). Each

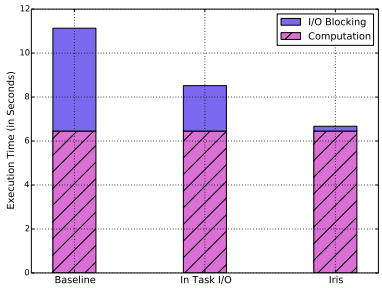


Figure 8: Performance of matrix multiplication. The application partitions an input matrix into 2048 sub-matrices of 256×256 double precision floats.

microbenchmark evaluates the performance of an individual optimization detailed in Section IV. In Section VI-D, we use S3D [10], a full combustion simulation, to evaluate the combination of all three optimizations focusing on how Iris reduces I/O-related overhead in a full application.

The experiments for all three microbenchmarks were conducted on a four node cluster. Each node has two sockets each with an Intel Xeon 5680 for a total of 12 physical cores per node (24 threads with hyperthreading), 48GB DRAM and a 250GB SSD drive. Nodes are connected with Mellanox QDR Infiniband. For all microbenchmarks, the Iris runtime dedicates 8 threads for computation and 4 threads for data transfers per node. The S3D experiments were performed on 64 nodes (i.e. 1024 cores) of the Titan supercomputer, a Cray XK7 system at Oak Ridge National Laboratory [11].

A. Matrix Multiplication

We use matrix multiplication to illustrate the effectiveness of deferred execution in accessing external resources. Our version of matrix multiplication uses matrices generated by an external application. The microbenchmark reads an input matrix from a disk file, partitions the entire matrix into same-sized sub-matrices, and multiplies every sub-matrix with an in-memory matrix individually.

Figure 8 illustrates the impact of deferred execution. In the baseline approach, we initially launch tasks that load the entire input matrix into a region. After the load phase, we launch tasks that perform matrix multiplication. The baseline shows how long the I/O and computation take, respectively.

A second approach simulates an application developer's efforts to overlap I/O with computation at the application level. The application performs in-task I/O at the beginning of each task, with the expectation that different tasks are executed at different speeds: some tasks can utilize the entire disk I/O bandwidth when other tasks are performing computation. This approach overlaps I/O with computation to some extent, with I/O blocking overhead reduced by 44% compared to baseline approach. However, the application still spends 24% of the execution time blocking on disk I/O.

In Iris, the application attaches the input matrix to a region. As discussed previously, by using attach/detach and acquire/release, the application allows Iris to manage disk I/O to maximize disk bandwidth and overlap I/O with

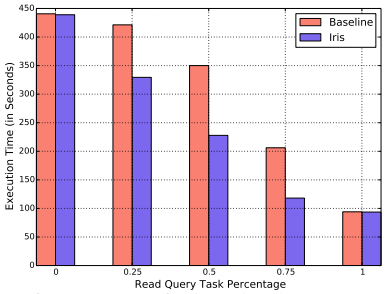


Figure 9: Performance of the database queries and updates benchmark. The application randomly generates 4096 database operations—the percentage of read queries relative to the entire set of operations is labeled on the x-axis.

computation. Compared to the first two approaches, Iris imposes negligible I/O blocking overhead.

B. Database Queries and Updates

The database query/update simulator, introduced in Section III, operates on a number of shared databases stored on disk. The application randomly generates query tasks (read-only) and update tasks (read-write); the specified order of the tasks is the order in which they are generated. We use this microbenchmark to measure the benefit of the write-after-read optimization.

The baseline approach attaches each database to an individual region and acquires the region before launching query or update tasks in Iris. As a result, all the benefit of deferred execution is exploited in the baseline approach. The write-after-read version also incorporates the write-after-read optimization described in Section IV-B.

Figure 9 shows the effect of the write-after-read optimization. We control the ratio between queries and updates to observe how write-after-read behaves in different workload distribution scenarios. When all operations are queries (i.e., $x = 1$ in the figure), the baseline and write-after-read achieve the same performance. In this scenario, there are no writes so the write-after-read optimization is not exercised but deferred execution still hides I/O latency and the lack of data dependencies makes the workload embarrassingly parallel. At the other extreme in which all operations are writes (i.e., $x = 0$) the two approaches yield the same performance as the write-after-read optimization is not exercised. The absolute performance for this workload (all writes) is lower as the write tasks on each database must be serialized to ensure coherence. At intermediate points (i.e., $0 < x < 1$), Iris is able to reduce the execution time by up to 43%, with the maximum benefit occurring around $x = .75$. The write-after-read optimization facilitates significantly more concurrency of tasks in this regime resulting in significant performance improvements.

C. Scene Rendering

Scene rendering, generating a pictorial representation of objects in the scene, is a standard application in computer graphics. Each object is a combination of properties such as shape, velocity, position, and texture. Our microbenchmark uses simple 2-D scenes, consisting of basic textured shapes (e.g., circles, triangles, and squares) moving in the scene.

All objects are managed by an external application residing on a single node (the master node), while the execution of the rendering application is distributed across multiple nodes. The objects may carry high-resolution texture/images making the efficiency of transferring the objects critical to the benchmark's overall performance. We use this rendering application to test the benefit of reducing data transfers as described in Section IV-C.

As objects in a scene may have different shape representations and different texture resolutions (i.e., various texture sizes) each object is presented to Iris as a pointer to an opaque data structure that holds all the information describing the object. The external application supports an interface for Iris tasks to call rendering methods, which include (1) rendering a portion of the background scene with the object; (2) modifying the properties of the object (e.g., velocity and position changes); and (3) the serialization and deserialization methods used by the Iris runtime to move the objects between different compute nodes.

Using these primitives we implemented a simple iteration-based rendering application. The application periodically re-renders the scene by assigning a portion of the background scene and all objects intersecting with that portion of the scene to a task. As different tasks are rendering disjoint parts of the scene, all tasks can be executed in parallel. The external application that stores and manages all objects is running on the master node, and every node (including the master node) renders a sub-scene.

The baseline approach attaches all objects to a 1-D region (the actual data type of the region is a pointer as described earlier) with each task acquiring a subset of objects to render the sub-scene. The optimized approach also reuses local copies when possible to reduce inter-node data transfers.

In the baseline approach, every task requires all objects that intersect with its sub-scene as an input. Without reducing data transfers all tasks fetch the objects from the master node. In the optimized approach, some tasks can directly access local copies of objects used by the previous timestep if the objects were not modified in the previous step. As Figure 10 illustrates, the ability to reduce data transfers results in significant performance improvements as inter-node data transfers begin to dominate total runtime at four nodes. Even though the two approaches spend the same amount of computation time in each iteration Iris reduces inter-node data transfers by up to 75%, significantly improving scalability and total performance.

D. S3D

S3D is a full application that performs direct numerical simulation of turbulent combustion. S3D was one of the

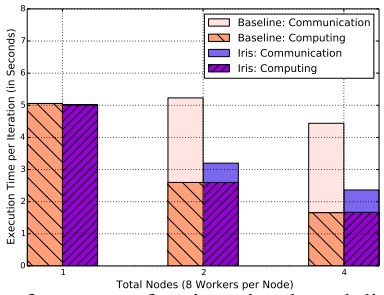


Figure 10: Performance of an iteration-based distributed rendering application. The microbenchmark loads 1024 objects from an external resource, with each object including 1-6MB of data representing the object.

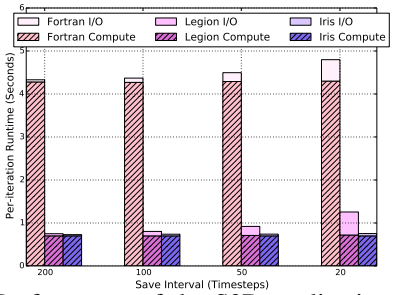


Figure 11: Performance of the S3D application running on 64 nodes (i.e., 1024 cores) of the Titan supercomputer.

six applications used to determine the Titan supercomputer's readiness for science applications. Direct numerical simulation attempts to resolve the smallest spatial and temporal features present in combustion, requiring a very large amount of state ($O(10^2 - 10^3)$ variables at each of $O(10^9)$ grid points). This state is much too large to save for every timestep, so scientists must decide how often to save snapshots of the state, balancing the I/O overhead against the risk of a system fault/interrupt that would necessitate restarting the application from a previous snapshot.

S3D was originally implemented in Fortran, but recent efforts have ported it to OpenACC[12] and to Legion[9] to take advantage of the heterogeneous processing power on Titan. The Iris version of S3D builds on top of the Legion implementation. Instead of explicitly performing file I/O to save snapshots within the Legion tasks, the Iris version attaches the disk files to regions and allows the runtime to perform the file I/O asynchronously from the main simulation loop, which can reduce (hide) I/O overheads and total runtime. For this experiment, we focus on the I/O overhead as a function of the snapshot interval, measuring it for the existing Fortran and Legion versions, as well as our Iris version. Results are illustrated in Figure 11.

For the Fortran version, a common choice is to snapshot every 100th timestep, which results in an I/O overhead of 0.1 seconds/iteration. The overhead increases linearly with the snapshot frequency, exceeding 0.5 seconds/iteration if every 20th timestep is snapshotted. The Legion version has a much faster simulation loop than the Fortran version but does nothing to improve the file I/O overheads. As is expected from Amdahl's Law, this causes the percentage overhead of file I/O to increase significantly relative to computation time. Even at a snapshot interval of 100 timesteps, the percentage overhead of file I/O is approximately 14%.

By performing the three optimizations described in Section IV, the Iris version achieves a relatively constant I/O overhead of 0.02-0.03 seconds/iteration from a 100 timestep interval to a 20 timestep interval. This shows that Iris's optimizations can efficiently hide I/O overheads and reduce total runtime. The Iris version of S3D nearly eliminates the dependence on the snapshot interval (up to bandwidth saturation of the file system), allowing scientists to adjust the snapshot interval based only on disk usage and system fault concerns and not the overhead of the I/O.

VII. Related Work

We summarize the most relevant work in integrating external resources with parallel systems.

A. Deferred Execution Parallel Systems

To the best of our knowledge, none of the existing parallel programming systems have defined semantics and mechanisms that allow the runtime system to manage and optimize external resource interaction with respect to computation. We summarize how existing parallel systems interact with external resources.

1) Parallel Systems for High-Performance Computing: Sequoia [2] is a portable runtime interface mapping data and computation to parallel machines with deep memory hierarchies. Sequoia includes the idea of using disk as a memory, but disk is modeled as internal storage rather than an external resource. Sequoia does not provide mechanisms for accessing external resources so applications must implement external I/O within tasks. In Iris, disk is used in two different ways: (1) it borrows Sequoia's idea of modeling disk as a large-capacity memory with low bandwidth and long latency, and (2) disk is also considered as an external resource that shares data with external systems.

MPI [13] provides integration with external resources through the MPI-IO interface. A number of optimizations have been developed in MPI-IO, including I/O aggregation [7]. While MPI-IO provides an asynchronous API for file I/O, effectively overlapping computation requires the application developer to identify work to overlap and schedule this work during the asynchronous I/O. Legion automates this overlap by scheduling independent tasks concurrently with I/O. Moreover, MPI-IO does not maintain consistency between external data in files and application-level copies in memory, and therefore misses optimizations such as reducing data transfer to/from external resources.

Charm++ [3] is a portable programming system for complex parallel applications. Charm++ is integrated with automatic checkpoint-based fault tolerance [14]. However, Charm++ does not support APIs for accessing external resources, so applications must explicitly perform in-task I/O to access external data and may miss Iris's optimizations, such as write-afer-read and reducing data transfers.

X10 [4] is a Java-based member of the PGAS family of languages. No special support for external resources is provided in X10, although as data objects and the places in which they reside are not migratable any data object in the system can in principle interface with an external resource.

2) Parallel Systems for Distributed Data Centers: Existing parallel systems for data centers (e.g., DryadLINQ [15], Spark [16], and MapReduce [17]) are well-integrated with on-disk storage systems. For example, all data objects in DryadLINQ are saved into distributed partitioned files, and therefore all LINQ queries must retrieve input from disk and write output to disk. This design persists intermediate results at the cost of introducing checkpoint overhead.

Most parallel systems for data centers use immutable data as their basic building block, which simplifies the implementation of failure recovery. Since read-only regions provide the same abstraction as immutable blocks of data, the ideas of Iris could also be applied to these systems.

B. I/O Middleware

I/O middleware libraries such as HDF5 [8] provide portable and extensible data models for managing persistent data. HDF5 is designed to support flexible and efficient I/O for complex data structures, which makes it an ideal toolkit for parallel systems with a hierarchical data model. HDF5 does not support deferred execution and does not maintain consistency between different copies of the same data. As a result, the optimizations discussed in this paper cannot be achieved solely within HDF5.

C. Memory Consistency

The idea of acquire and release operations in Iris is borrowed from the *release consistency* memory model proposed by Gharachorloo et al. [5]. Release consistency requires that all writes to memory before a release operation must be visible to all reads after a subsequent acquire operation. In Iris, the acquire/release operations provide this semantics.

VIII. Conclusion

We have presented Iris, a task-based deferred execution model with external resources. Iris defines semantics for external resources and provides a number of optimizations for reducing external I/O overhead. We have presented an implementation, which is built on Legion and includes all the optimizations discussed in the paper. Finally, we have evaluated Iris on three microbenchmarks and a full combustion simulation.

Acknowledgments

This work was supported by Los Alamos National Laboratories Subcontract No. 173315-1 through the U.S. Department of Energy under Contract No. DE-AC52-06NA25396 as well as DoE Office of Science grant DE-SC0012426. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC'12*, Salt Lake City, Utah, 2012.

[2] M. Houston, J.-Y. Park, M. Ren, T. Knight, K. Fatahalian, A. Aiken, W. Dally, and P. Hanrahan, "A portable runtime interface for multi-level memory hierarchies," in *PPoPP'08*, Salt Lake City, UT, 2008.

[3] L. V. Kale and S. Krishnan, "CHARM++: a portable concurrent object oriented system based on C++," in *OOPSLA'93*, New York, NY, 1993.

[4] 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 *OOPSLA'05*, San Diego, CA, 2005.

[5] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *ISCA'90*, New York, NY, 1990.

[6] 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*, vol. 26, no. 7, 2014.

[7] R. Thakur, W. Gropp, and E. Lusk, "Data sieving and collective I/O in romio," in *Frontiers'99*, 1999.

[8] "Introduction to HDF5," https://www.hdfgroup.org/HDF5/doc/H5.intro.html.

[9] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Structure slicing: Extending logical regions with fields," in *SC'14*, New Orleans, LA, 2014.

[10] 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. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo, "Terascale direct numerical simulations of turbulent combustion using S3D," *Computational Science and Discovery*, vol. 2, no. 1, 2009.

[11] A. S. Bland, J. C. Wells, O. E. Messer, O. R. Hernandez, and J. H. Rogers, "Titan: Early experience with the Cray XK6 at Oak Ridge National Laboratory," in *CUG'12*, Stuttgart, Germany, 2012.

[12] J. Levesque, R. Sankaran, and R. Grout, "Hybridizing S3D into an exascale application using OpenACC: An approach for moving to multi-petaflops and beyond," in *SC'12*, Salt Lake City, UT, 2012.

[13] "The Message-Passing Interface," http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[14] G. Zheng, C. Huang, and L. V. Kalé, "Performance evaluation of automatic checkpoint-based fault tolerance for AMPI and Charm++," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 2, 2006.

[15] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *OSDI'08*, San Diego, CA, 2008.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI'12*, San Jose, CA, 2008.

[17] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04*, Berkeley, CA, 2004.