TaskWorks: A Task Engine for Empowering Asynchronous Operations in HPC Applications

Kaiyuan Hou*, Quincey Koziol†, Suren Byna†
*Northwestern University — {khl7265}@eecs.northwestern.edu
† Lawrence Berkeley National Laboratory — {koziol,sbyna}@lbl.gov

I. Background

Communication and I/O operations can consume much of the overall execution time of High-Performance Computing (HPC) applications. Overlapping those operations with computation can improve application performance and resource utilization. To reap these benefits, many HPC libraries provide APIs for an application to execute some operations asynchronously. The application can then continue with computations until it needs the result of the asynchronous operations.

Application operations can depend on each other. These dependencies can be fulfilled by performing the operations synchronously in the correct order. However, this is not easy to achieve with asynchronous operations. A task engine must ensure that dependencies are fulfilled for asynchronous operations as well. For example, among I/O operations in HDF5 [1], [2], a dataset write operation can only start after the dataset itself is opened. Similarly, an open HDF5 group cannot be closed until all operations that create or open objects within the group have been completed. (See *Figure 1* in the poster for an illustration of these dependencies)

Most asynchronous operations execute on a background thread. Programming with low-level thread APIs such as pthreads is tedious and error-prone, especially when there are dependencies among operations that require ordering or synchronization. However, many high-level task libraries have a complex API and programming model that takes effort to learn and adopt. Some of them also bear the overhead of supporting features not needed for asynchronous operations, and their support for task dependencies is usually limited to basic forms that only run a dependent task when all parent tasks complete. Finally, many task libraries are designed for a serial execution environment and do not consider the interactions between compute nodes that occur in an HPC environment. (See *Table 1* in the poster for a comparison of task libraries)

II. Design goals

To empower asynchronous operations in HPC applications, we introduce TaskWorks – a high-level, light-weight task execution engine. The design of TaskWorks has four goals:

User-friendly: The purpose of TaskWorks is to create a light-weight task engine that hides the complexity of lower-level thread APIs from application developers. TaskWorks provides an easy-to-use *post-and-forget* API for executing operations in HPC applications asynchronously. It provides an interface to add asynchronous capabilities to existing libraries with little effort from a developer.

Portable and extendable: HPC platforms are diverse, and their hardware and runtime environment can vary significantly from one system to another. TaskWorks contains pluggable abstraction layers internally, which allow it to use a variety of low-level thread and task manager back-ends. It can run on top of low-level thread packages such as pthreads and Win32 threads as well as high-level libraries such as Argobots [3] and Kokkos [4]. Extensions can also be created to adapt the framework to other libraries.

Flexible: TaskWorks makes it easy to perform simple tasks, such as running asynchronous operations, while also providing support for more advanced usage. An example of this design principle is the support of custom task dependencies and threadless (polled) task execution. This allows TaskWorks to be used in scenarios beyond just executing asynchronous operations.

HPC-friendly: TaskWorks is designed specifically for HPC environments. A TaskWorks task may use MPI to communicate with another task on a different compute node. These tasks should be run at the same time to prevent blocking or deadlocks. Some tasks can also compete for a shared resource, such as the file system. TaskWorks has mechanisms to avoid scheduling them at the same time to reduce contention. Finally, an application can use multiple thread/task packages at the same time. For example, the application may run OpenMP alongside TaskWorks or within tasks. In addition to being compatible with those packages, TaskWorks must coordinate its resource usage with them to prevent overloading the system.

III. architecture

TaskWorks's programming model revolves around three object types: tasks, events, and engines. Tasks are operations that can run asynchronously, possibly with dependencies on other tasks or events. Events monitor objects, such as files, sockets, timers, and MPI operations, and execute an operation when a condition is met. Engines contain and organize tasks and events. See Fig. 1 for an overview of TaskWorks's architecture.

Engines manage a pool of threads that can execute task and event operations in the background. The number of threads can be adjusted dynamically, and an engine can also be created without any threads. In a threadless scenario, an engine serves as a task manager that coordinates dependencies and is polled by the application to make progress on tasks using the calling

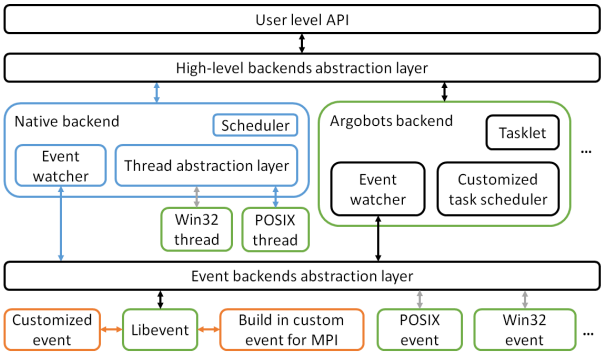

Fig. 1: TaskWorks Architecture Overview

thread for a user-specified time period. Multiple engines can also run in parallel, allowing a different number of threads to be dedicated to different types of tasks.

Threads in an engine are called *workers* and are shared by tasks and events assigned to the engine. Workers loop on the main scheduler routine until the engine is freed. In each iteration, workers check the event loop for any pending events and execute the event handler for every pending event. Events are always prioritized over tasks. When there are no active events left, workers check for tasks. Should there be no tasks as well, the workers will sleep for a period of time or until signaled before starting another scheduling iteration. Although events have priority over tasks, they can still be delayed if there are no available workers in the pool. (*Figure 3* in the poster illustrates the workflow of the main scheduler.)

TaskWorks allows three priority levels for user tasks - high, standard, and lazy, with a separate queue for each priority level. When a task becomes ready to run, it is inserted into the queue that corresponds to its priority. There are also reserved priority levels for internal use, such as running the event handler. When looking for tasks, a worker checks the queues in the order of priority and takes the first task it gets. The lazy queue is only checked when there is more than one idle worker thread, guaranteeing that a task in the lazy queue will never delay a task in a higher queue.

Tasks are the basic work units in TaskWorks. Applications can submit an operation to execute asynchronously by creating a task and assigning it to an engine. A task contains a user function to run, called the *handler*, its arguments, and other metadata. Tasks created without a handler are marked complete when they become ready to run and can serve as a place holder for other tasks to depend on. For example, the predefined *barrier task* is marked complete when all tasks created before it complete.

To control task execution ordering, applications can set dependencies for tasks. A task can depend on tasks handled by another engine in the same process, or even an engine on another MPI rank. Every task maintains a list of tasks it depends on and a list of tasks that depend on it. The dependency list in each engine forms a workflow graph in which tasks are nodes, and dependencies are edges. Many other high-level task libraries require this workflow graph to be a Directed Acyclic Graph (DAG) and require all depended-on tasks to complete before a task can start. TaskWorks is more flexible about its task dependencies and allows users to define custom task dependency functions that determine whether a task can start, based on the status of the tasks it depends on. Since the dependency function can allow a task to start before the tasks it depends on complete, a cycle in the workflow graph does not necessarily result in a deadlock.

Applications start a task by committing (assigning) it to an engine. If a task has no dependencies, it is pushed directly into the job queue when committed. Otherwise, it is inserted into the dependent list of all tasks it depends on. When the status of a depending task changes, such as when they complete, dependent tasks are notified by calling their dependency callback functions. Should the dependency function indicate that a task is ready to run, it is pushed into the engine's job queue.

TaskWorks has an integrated event handler and can monitor file, socket, MPI, timer, and task-related events. Events can interact with tasks and vice-versa. Applications can create or manage tasks inside the event handler. Tasks also trigger events when their status change.

Internally, TaskWorks contains abstraction layers that allow it to work with various backends. There is a pluggable layer immediately below the user-level API so that multiple implementations of TaskWorks functions are possible through plugin modules. This allows TaskWorks to be built on top of existing high-level task libraries. TaskWorks also contains a default plugin that implements TaskWorks APIs directly on top of low-level thread libraries. Inside this is another abstraction layer to allow portability between low-level thread APIs on different platforms. A third abstraction layer provides a unified event interface for high-level modules. TaskWorks's modular design allows it to employ a wide variety of external libraries at both high and low levels.

TaskWorks coordinates with other thread/task execution packages, such as OpenMP or MPI, through the PMIx interface [5]. PMIx allows TaskWorks to communicate its resource usage with other packages running on the system and allows it to take proper actions when notified of their resource needs.

IV. Conclusion

In this poster, we present the need for TaskWorks, and its design goals and architecture. The development of TaskWorks is in progress: we have completed the implementation of Argobots backend (in Fig. 1), and the development of the native backend is in progress.

References

[1] "The HDF5® library & file format." [Online]. Available: https://www.hdfgroup.org/solutions/hdf5/

[2] M. Folk *et al.*, "An overview of the HDF5 technology suite and its applications," in *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, 2011, pp. 36–47.

[3] S. Seo *et al.*, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2017.

[4] H. Edwards *et al.*, "Kokkos: Enabling performance portability across manycore architectures," in *2013 Extreme Scaling Workshop (xsw 2013)*. IEEE, 2013, pp. 18–24.

[5] R. H. Castain *et al.*, "PMIx: process management for exascale environments," *Parallel Computing*, vol. 79, pp. 9–29, 2018.