The Design and Implementation of AXE, an Asynchronous Execution Engine

Chao Mei, Quincey Koziol, Dana Robinson, Neil Fortner, Ruth Aydt

This document describes the interface and operation of AXE, an asynchronous execution engine that can be used to asynchronously execute user-defined functions that have execution order dependencies. A high-level design of the AXE implementation is also presented.

1. Introduction

Asynchronous execution can improve application performance by overlapping computation with other operations, such as network communication and file I/O. For example, consider a scenario where an I/O function call to create a file on a remote file server is followed by computation. With synchronous execution, the I/O function call will not return until it finishes creating the file on the remote file server. If the computation is not dependent on the file creation, then the computation can be overlapped with the (potentially slow) I/O function call. To enable such overlap, the I/O function call is made asynchronous[1].

Expanding the example further, suppose the operations to be performed are "create file", "write data to file", and "perform computation". In this case, "create file" must finish before "write data to file" can begin, while "perform computation", which is independent of the I/O function calls, can overlap with both. The create and write function calls can be executed asynchronously, but they have an execution order dependency which must be honored.

To support asynchronous execution of functions that can have execution order dependencies, we are developing an "Asynchronous eXecution Engine" called *AXE*. The goal of AXE is to provide:

- A rich and intuitive interface for specifying operations and their dependency relationships
- An efficient engine that asynchronously executes operations when all operations it depends on have completed and compute resources are available
- A mechanism for monitoring execution status and results, and for facilitating data sharing across multiple asynchronous operations

Developers can use AXE to move from synchronous to asynchronous operations in existing programs, or to write new programs with asynchronous execution paths. We believe that AXE will reduce the burden of managing multiple dependent asynchronous functions and thread pools, leading to increased programmer productivity and greater adoption of asynchronous programming.

[1] In this document, an asynchronous function call is assumed to also be non-blocking.

2. Motivation and Related Work

Our ongoing efforts to improve the performance of HDF5 in increasingly multi-core and multi-layered environments have pointed to heavier reliance on asynchronous function calls with execution-order dependencies. In many cases, not only must one call complete before another can begin, but the second call needs results from the first. Consider the "create file" / "write data to file" operations in the earlier example. Not only must the file be (successfully) created before the write can occur, but the write call needs the file handle returned by the create call. Managing execution order, return codes, and shared data can be quite complex.

Rather than coding the sequenced asynchronous calls on a case-by-case basis, we have opted to develop AXE, an Asynchronous eXecution Engine. AXE allows the user (programmer) to: (1) define functions that implement operations be executed asynchronously, (2) define data structures that allow data to be passed and shared across asynchronous function calls, (3) specify execution-order dependencies across multiple asynchronous functions, (4) have the functions executed asynchronously in a sequence that honors the declared dependencies, and (5) check on function execution status. In many cases, the new user-defined functions can merely be wrappers for existing functions that handle the data passing and sharing.

The idea of specifying and managing multiple tasks that depend on each other is not a new one, and we have drawn ideas from this prior work. DAGman[2], provides task-scheduling capability at a program-level-basis on distributed, unrelated compute resources. With DAGman, the order of execution is specified, and static, before the main job is submitted. More recently, DAGue[3] has been released for numerical algorithm tasks. Both show the value of a framework for managing dependent tasks, but neither provide the granularity or flexibility we need for our work.

3. The Dependence Graph and Asynchronous Execution

In this section, we discuss directed acyclic graphs (DAGs) and the particular characteristics of DAGs as they are used to specify asynchronous execution in AXE.

3.1 Directed Acyclic Graphs

As demonstrated by DAGman, DAGue, and other related work, the execution order of a set of tasks can be represented by a directed acyclic graph (DAG). A task is represented by a node in the DAG, and a dependence relationship is represented by a directed link between two nodes.

Figure 1 shows a very simple DAG with two tasks, T1 and T2, represented by the two nodes in the graph. The directed link between the nodes indicates that T1 must complete before T2 can start.

[2] http://research.cs.wisc.edu/htcondor/manual/v7.6/2_10DAGMan_Applications.html
[3] Need bibref for DAGue

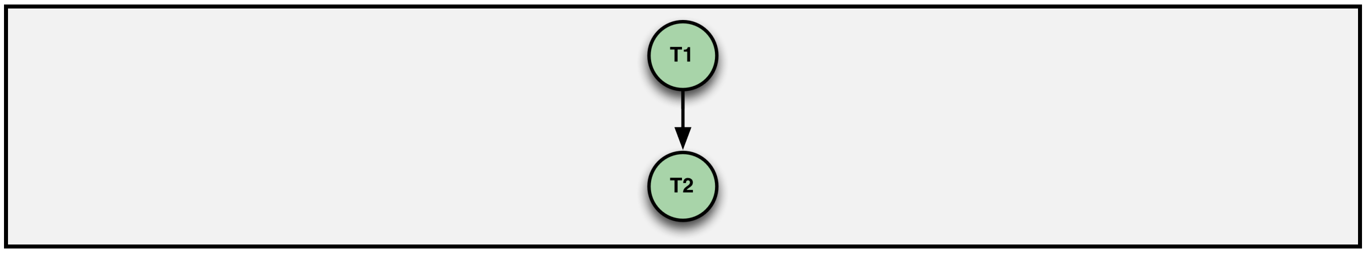

Figure 1: A simple Directed Acyclic Graph (DAG).

When used with AXE, T1 and T2 in Figure 1 both represent asynchronous tasks from the application's perspective. Referring again to the "create file" / "write data to file" / "perform computation" example, T1 in Figure 1 corresponds to the "create file" operation and T2 corresponds to "write data to file" operation. While AXE orchestrates the asynchronous execution of T1 and T2, the application can perform computation, checking back later to find the status of the create and write calls.

3.2 Dependence Flow Graphs

Figure 2 and **Error! Reference source not found.** show more complex DAGs that demonstrates the variety of dependence relationships supported by AXE. We refer to a DAG that defines the execution order dependencies for asynchronous tasks to be executed by AXE as a *dependence flow graph*.

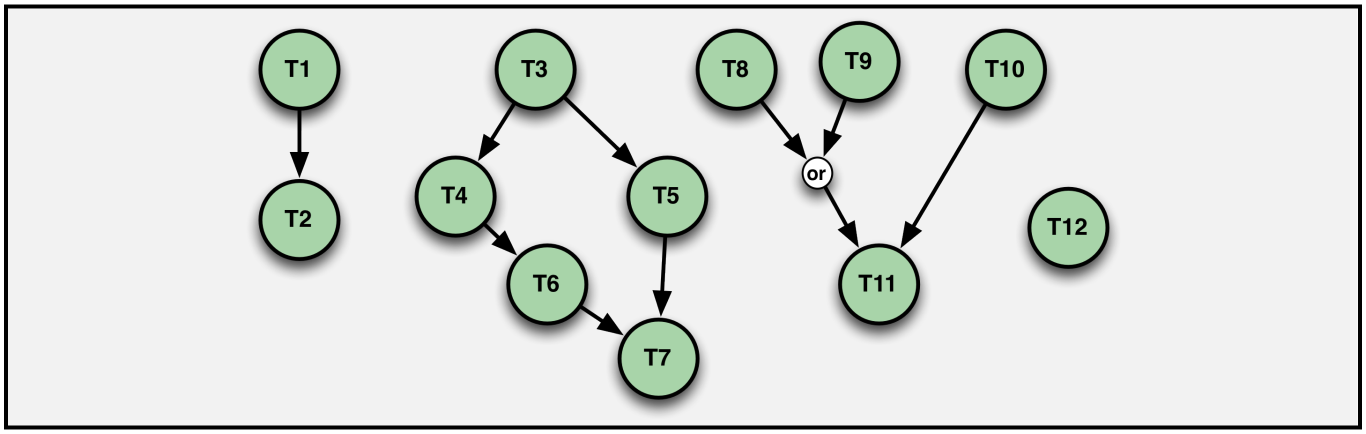

Figure 2: An AXE dependence flow graph.

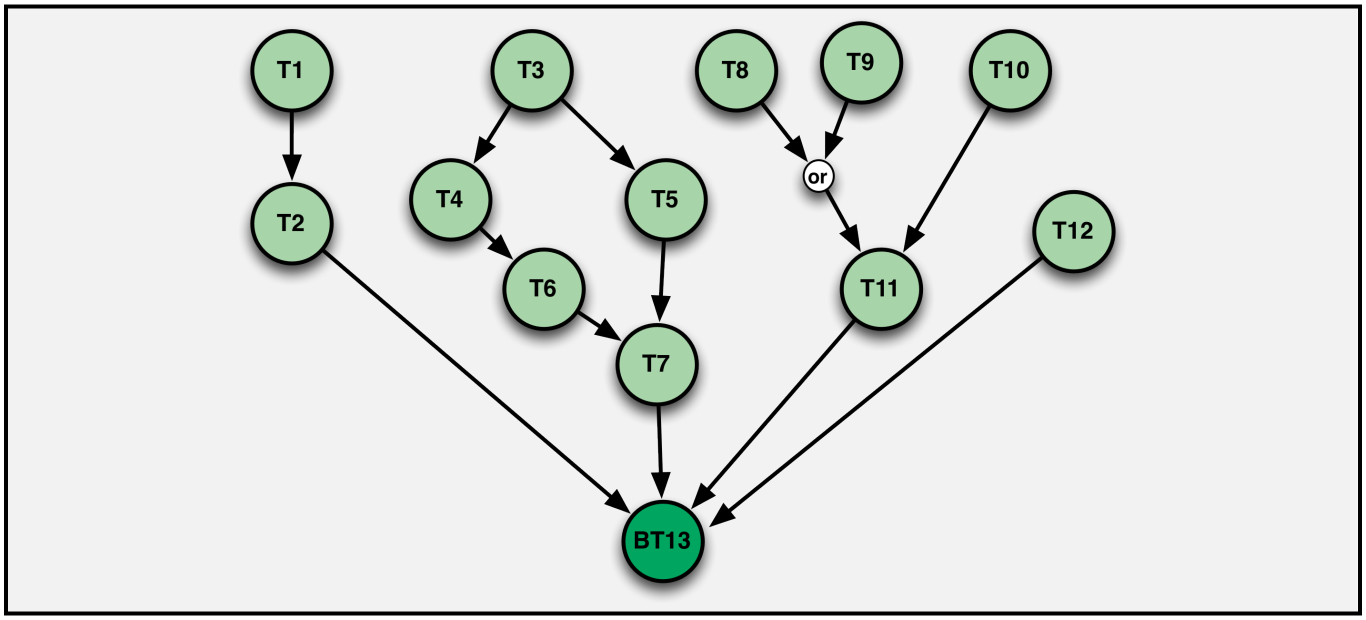

Figure 3: The AXE dependence flow graph from Figure 2 after a barrier task has been added.

AXE provides APIs to construct dependence flow graphs and allows new tasks to be added to a graph at any time—even after execution is underway. An overview of working with AXE, including graph construction, is given in Section 4. Section 5 covers the AXE APIs in detail.

3.2.1 Dependence Flow Graph Terminology

AXE dependence flow graph terminology and characteristics, with concrete examples drawn from Figure 2 and **Error! Reference source not found.**, are presented here:

- Task X is a *parent* of task Y if the execution of task Y depends on the completion of task X.
- Task Y is a *child* of task X if the execution of task Y depends on the completion of task X.
- A task can have multiple parent tasks.
- A task can have multiple child tasks.
    - T1 is parent of T2; T3 is parent of T4 and T5; T4 is parent of T6; T5 and T6 are parents of T7; T8, T9, and T10 are parents of T11. In Figure 3, T2, T7, T11, and T12 are parents of BT13.
    - T2 is child of T1; T4 and T5 are children of T3; T6 is child of T4; T7 is child of T6 and T5; T11 is child of T8, T9, and T10. In Figure 3, BT13 is a child of T2, T7, T11, and T12.
- A task with no parents is called a *root task.* There may be multiple root tasks in a dependence flow graph.
    - T1, T3, T8, T9, T10, and T12 are root tasks.
- A task with no children is called a *leaf task.*
    - T2, T7, T11, and T12 are leaf tasks in Figure 2.
    - BT13 is the leaf task in Figure 3.
- All tasks inserted into an AXE engine will eventually execute, as their dependencies are fulfilled, unless canceled by the application.

- A child task has a *necessary* dependence relationship with a parent task if the parent task must complete before the child task can execute.
    - The necessary dependence relationships are T2 with T1; T4 with T3; T5 with T3; T6 with T4; T7 with T6; T7 with T5; T11 with T10. In Figure 3, there are these additional necessary dependence relationships: BT13 with T2; BT13 with T7; BT13 with T11, and BT13 with T12.
- A child task has a *sufficient* dependence relationship with a set of parent tasks if only one of the parent tasks in the set must complete before the child task can execute. Although the child's execution depends on only one of the parent tasks completing, all parent tasks will execute (as their own dependencies are fulfilled). A child task can have a sufficient dependence relationship with no more than one set of parent tasks.
    - The *sufficient* dependence relationship is T11 with {T8, T9}
- A *barrier* task is a special kind of task that acts as a synchronization point for other tasks. In AXE dependence flow graphs, all leaf tasks in the graph at the time the barrier task is added become parents of the barrier. The barrier task has necessary dependence relationships with each of its parents. Tasks added to the graph after a barrier task may have a necessary or sufficient dependence on the barrier, or no dependence on it at all.
    - Figure 3 shows the dependence flow graph that results when a barrier task is added to the graph shown in Figure 2

3.2.2 Dependence Flow Graph Task Execution Order

Tasks in an AXE dependence flow graph follow an execution order that is constrained by their relationships to other tasks in the graph. The rules governing execution order, with concrete examples drawn from Figure 2 and Figure 3, are given here:

- Before a task can execute, all of its necessary parent tasks and at least one of its sufficient parent tasks must complete.
    - Before T11 can execute, T10 and either T8 or T9 must complete.
- The execution order of the asynchronous tasks follows the dependence relationships defined by the directed links in the AXE dependence flow graph.
    - T3 will execute before T4; T4 will execute before T6; T6 will execute before T7. And, following the same dependence path in Figure 3, T7 will execute before BT13.
- Tasks without any dependence relationship (those that do not appear on the any path of directed links) may execute in any order, including concurrently.
    - T3 may execute before, after, or concurrent with T1, T2, T8, T9, T10, T11, and T12.
    - T4 may execute before, after, or concurrent with T5 and all other tasks in Figure 2 except T3, T6, and T7.
    - T4 may execute before, after, or concurrent with T5 and all other tasks in Figure 3 except except T3, T6, T7, and BT13.
    - T12 may execute before, after, or concurrent with all other tasks in Figure 2.
    - T12 may execute before, after, or concurrent with all other tasks in Figure 3 except BT13.
- When a child task has a sufficient dependence relationship with a set of parent tasks, there is no strict order of execution of the parent tasks and the child task, other than one parent task must execute before the child task.

- Tasks T8, T9, and T11 may execute in the following order relative to each other. Square brackets are used to show tasks executing concurrently. The concurrently executing tasks may not have different start and end times, that is only partial concurrent execution:
    - T8, T9, T11
    - T9, T8, T11
    - T8, T11, T9
    - T9, T11, T8
    - [T8, T9], [T11, T8] // T9 finishes before T8, so T8 keeps running
    - [T8, T9], [T11, T9] // T8 finishes before T9, so T9 keeps running

4. Working with AXE

In this section, we discuss how to work with AXE at a conceptual level.

At the highest level, an application creates an AXE engine and submits tasks to the engine, building a dependency flow graph. The engine uses the graph to determine what task(s) to execute next. AXE is implemented as a library that is linked with an application and uses an internal thread pool for each engine to execute tasks in that engine's graph.

Using AXEcreate_engine, an application creates an AXE engine[4] and gets a handle for the engine. AXE engines have a number of tunable attributes, such as the number of threads in their thread pool, the ID range, etc. which can be set by the application before creating the engine[5].

Then the application inserts new tasks into the engine, possibly with lists of other tasks that are necessary or sufficient dependencies for each new task, using AXEcreate_task or AXEcreate_barrier_task. An AXE task is made up of several distinct components:

- A task ID value, which can be either provided by the application or generated by AXE (before creating the task)
- A pointer to a user-defined operation function that will be executed when the task is executed
- A pointer to a user-managed data buffer used for the user-defines function's parameters and return value
- A pointer to a user-defined free function that will be invoked by the AXE engine to free the used-managed data buffer

When each task's necessary and sufficient parent conditions are met, the task's operation function is executed by AXE.

Tasks executed by AXE can modify the task DAG during the operation function, including adding new tasks or removing existing ones. Tasks executed by AXE may also retrieve the user-managed data buffer for other tasks. This capability is intended to allow child tasks to retrieve information from

[4] While there is no limitation on the number of engines a single application can create, we restrict the discussion to a single engine for simplicity. Each AXE engine operates independently of all others, so there is simplification is a reasonable one.

[5] See `AXEset_num_threads`, `AXEset_id_range`, `AXEset_num_id_buckets`, `AXEset_num_id_mutexes` below.

parent tasks, allowing information to be passed through the dependency chain. Each child task must indicate to AXE that it is finished with its parent task's user-managed data buffer by calling one of the finish routines (AXEfinish orAXEfinish_all) on each parent task's ID. When all child tasks are finished with a parent task's user-managed data buffer, AXE invokes the task's free function to free the buffer.

5. AXE Type Definitions and APIs

In this section, we show the set of type definitions and APIs that together provide the application interface to the AXE library.

5.1 Type Definitions and Constants

5.1.1 AXE_engine_t

```
typedef AXE_engine_int_t *AXE_engine_t;
```

A handle for an asynchronous execution engine – an AXE. The underlying type (AXE_engine_int_t) is not for application inspection.

5.1.2 AXE_engine_attr_t

```
typedef struct {
   size_t num_threads;
   size_t num_buckets;
   size_t num_mutexes;
   AXE_task_t min_id;
   AXE_task_t max_id;
} AXE_engine_attr_t;
```

AXE_engine_attr_t contains fields for the tunable aspects of an AXE engine. The fields are defined as:

- num_threads: The number of threads used for tasks execution by the engine
- num_buckets: The number of buckets in the task ID table, which is a simple hash table with a function of (id % num_buckets)
- num_mutexes: The number of bucket mutexes in the ID table
- min_id: The minimum ID for automatically generated IDs
- max_id: The maximum ID for automatically generated IDs

This structure should be initialized with AXEengine_attr_init, the fields set with AXEset_<property>, and the structure finalized with AXEengine_attr_destroy.

5.1.3 AXE_task_t

```
typedef uint64_t AXE_task_t;
```

An asynchronous task ID value in AXE.

5.1.4 AXE_error_t; AXE_SUCCEED, AXE_FAIL

```
typedef enum {
   AXE_SUCCEED,
   AXE_FAIL
} AXE_error_t;
```

AXE_error_t is the type returned by AXE API calls.

All AXE APIs return AXE_SUCCEED on success and AXE_FAIL on failure.

5.1.5 AXE_status_t; AXE_TASK_WAITING_FOR_PARENT, AXE_TASK_SCHEDULED, AXE_TASK_RUNNING, AXE_TASK_DONE, AXE_TASK_CANCELED

```
typedef enum {
   AXE_TASK_WAITING_FOR_PARENT,
   AXE_TASK_SCHEDULED,
   AXE_TASK_RUNNING,
   AXE_TASK_DONE,
   AXE_TASK_CANCELED
} AXE_status_t;
```

AXE_status_t is the type used to report the status of a task in AXE.

The five possible states are: 1) waiting for one or more parent tasks, 2) ready to execute when thread available, 3) executing, 4) finished, and 5) canceled.

5.1.6 AXE_remove_status_t; AXE_CANCELED, AXE_NOT_CANCELED, AXE_ALL_DONE

```
typedef enum {
   AXE_CANCELED,
   AXE_NOT_CANCELED,
   AXE_ALL_DONE,
} AXE_remove_status_t;
```

AXE_remove_status_t is the type used to report the status of removing a task with AXEremove or AXEremove_all.

The three possible states are: 1) all tasks were canceled or already complete, 2) at least one task was not canceled because it was running, 3) all tasks were already complete.

5.1.7 AXE_task_op_t

```
typedef void (*AXE_task_op_t)(AXE_engine_t engine,

   size_t num_necessary_parents, AXE_task_t necessary_parents[],
   size_t num_sufficient_parents, AXE_task_t sufficient_parents[],
   void *op_data);
```

AXE_task_op_t is the function pointer type for a user-defined function that is submitted to AXE for asynchronous execution. The memory for the *op_data* parameter (which includes both the operation's parameters and its return value) must be managed by the user. When an operation is invoked by the execution engine, the engine passes arrays of task handles to the necessary and sufficient parent tasks that have completed and have allow this operation to be invoked. The array of sufficient parent task handles that is passed in to this callback will only contain those sufficient parent tasks that have completed, and may not include all of the sufficient parent task handles used in the creation call for the task. The operation must call AXE_finish or AXE_finish_all on these task handles, after retrieving any information about their tasks (*e.g.: op_data*) that it requires.

The operation routine will only be invoked once for each task, and has no return value.

5.1.8 AXE_task_free_op_data_t

typedef void (*AXE_task_free_op_data_t)(void *op_data);

This function pointer type refers to a routine that is invoked by the engine when the ‘finish’ API call has been made on all outstanding references of a task. It is intended to be used to release the *op_data* buffer, and the Standard C *free()* routine can simply be passed if that is sufficient.

5.1.9 AXE API routines

All routines below return AXE_error_t to indicate success or failure of the operation.

APIs for engine attribute initialization, tuning and termination:

a) **AXEengine_attr_init(/*OUT*/ AXE_engine_attr_t *attr)**

   This function initializes an instance of an engine attribute structure, setting default values. Must be called prior to use of the engine attribute structure.

   Engine attributes are not currently meant to be accessed concurrently by multiple threads unless they are not modified while being used concurrently. If the application wants to modify an attribute while it is being used by another thread, it is the application's responsibility to protect the attribute with a mutex.

b) **AXEengine_attr_destroy(AXE_engine_attr_t *attr)**

   This function destroys an instance of an engine attribute structure , freeing any memory used in any internal fields. Currently does nothing, but may be necessary in a future version.

c) **AXEset_num_threads(AXE_engine_attr_t *attr, size_t num_threads)**

   AXEget_num_threads(const AXE_engine_attr_t *attr, /* OUT */size_t *num_threads)

   These functions set/get the number of threads to use for an engine. An engine created using this attribute will always create and maintain *num_threads* threads, with no extra threads set aside to, for example, schedule tasks. All task scheduling is done on-demand by application threads (in AXEcreate_task and AXEcreate_barrier_task) and by the task execution threads.

   It is guaranteed that there will always be exactly *num_threads* available for execution (with the possibility of a short wait if the thread is performing its scheduling duties), until AXEterminate_engine is called. Therefore, if the application algorithm requires N tasks to run concurrently in order to proceed, it is safe to set *num_threads* to N.

   The default value is 8. (This could be added as a configure option in a future version.)

d) **AXEset_id_range(AXE_engine_attr_t *attr, AXE_task_t min_id, AXE_task_t max_id)**

AXEget_id_range(const AXE_engine_attr_t *attr, /* OUT */ AXE_task_t *min_id, /* OUT */ AXE_task_t *max_id)

These functions set/get the minimum and maximum task ID values for automatically generated task IDs (with AXEgenerate_task_id) for an engine. All IDs generated by AXEgenerate_task_id will fall between min_id and max_id (inclusive). If all IDs in the range are in use, AXEgenerate_task_id will fail.

Care must be taken if mixing use of AXEgenerate_task_id with manual assignment of task IDs, as it is possible that AXEgenerate_task_id could generate an ID that the application thinks is free. It is probably a good idea to manually assign IDs from outside the range specified in this function.

The default values are 0 and UINT64_MAX.

e) **AXEset_num_id_buckets(AXE_engine_attr_t *attr, size_t num_buckets)**

AXEget_num_id_buckets(const AXE_engine_attr_t *attr, /* OUT */ size_t *num_buckets)

These functions set/get the number of buckets in the engine's id hash table on the provided engine attribute. The hash function is simply ID % *num_buckets*.

Setting a larger value for *num_buckets* will reduce the number of hash value collisions, improving performance in most cases, but will make iteration more expensive, reducing performance for AXEcreate_engine, AXEterminate_engine, AXEcreate_barrier_task, and AXEremove_all, and will consume more memory. Setting a smaller value for *num_buckets* will have the opposite effect.

The default value is 10007.

f) **AXEset_num_id_mutexes(AXE_engine_attr_t *attr, size_t num_mutexes)**

AXEget_num_id_mutexes (const AXE_engine_attr_t *attr, /* OUT */ size_t *num_mutexes)

These functions set/get the number of mutexes used to protect ID hash buckets on the provided engine attribute. Each hash bucket uses its mutex to prevent simultaneous to elements in the bucket (except in some cases where it is allowed). The index of the mutex used by a hash bucket is given by *bucket_index* % *num_mutexes*.

Setting a larger value for *num_mutexes* will reduce contention for the mutexes, improving performance, but will consume more memory. Setting a smaller value for *num_mutexes* will have the opposite effect. There is no benefit to setting *num_mutexes* to a value larger than *num_buckets*.

The default value is 503.

APIs for engine creation and termination:

a) **AXEcreate_engine(/*OUT*/ AXE_engine_t *engine, const AXE_engine_attr_t *attr)**

This function creates and initializes an instance of an engine for asynchronous execution, with the attributes for the engine specified by *attr* (if present) or default (if NULL). The handle for the newly created engine is returned in **engine*.

b) **AXEterminate_engine(AXE_engine_t engine, int wait_all)**

This function terminates an instance of the execution engine.

The engine will first deal with uncompleted tasks. If the function parameter *wait_all* is set to true (non-zero), then this function blocks the program execution until all uncompleted tasks complete. Otherwise, uncompleted tasks will be aborted. Afterwards, all allocated resources will be released and the engine instance terminates.

Any concurrent or subsequent use of this engine or the tasks within it is an error and will result in undefined behavior. An exception to this rule is made if *wait_all* is true, in which case the engine and tasks may be manipulated as normal while tasks are still running. It is the application's responsibility to make sure that tasks are still running. It is safe to call AXEwait in this case, even if the task waited on may be the last task to complete (again as long as tasks are still running when calling AXEwait).

APIs for managing task IDs:

a) **AXEgenerate_task_id(AXE_engine_t engine, /* OUT */ AXE_task_t *task_id)**

Generate a task ID that can be used in a subsequent call to AXEcreate_task or AXEcreate_barrier_task. The ID will be between the *min_id* and *max_id* parameters given to AXEengine_attr_id_range. The ID is returned in **task*. The ID must never be reused by an application, even if AXEfinish or AXEfinish_all is called on the task, unless it is returned by another call to AXEgenerate_task_id. If the ID is no longer needed and has not been used it can be released with AXErelease_task_id.

b) **AXErelease_task_id(AXE_engine_t engine, AXE_task_t task_id)**

Release the specified task ID back to the engine, allowing it to be reused. The task must have been previously generated by AXEgenerate_task_id, and must not have been used to create a task.

APIs for creating/destroying asynchronous tasks:

a) **AXEcreate_task(AXE_engine_t engine, AXE_task_t task_id,**

size_t num_necessary_parents, AXE_task_t necessary_parents[],

size_t num_sufficient_parents, AXE_task_t sufficient_parents[],

AXE_task_op_t op, void *op_data, AXE_task_free_op_data_t free_op_data)

This function creates a new task in the engine identified by *engine*, setting its operation routine to *op* with parameter *op_data*. This task may depend on other tasks, specified in the *necessary_parents* array of size *num_necessary_parents* and the *sufficient_parents* array of size *num_sufficient_parents*. All tasks in the *necessary_parents* array must complete and at least one of the tasks in the *sufficient_parents* array must complete before the engine will execute this task, but either of the necessary or sufficient sets of parent tasks (or both) can be empty.

If any parent tasks complete before the new task is inserted, they will still be included in the arrays passed to the *op* routine when it is invoked. If the *op* parameter is NULL, no operation will be invoked and this event is solely a placeholder in the graph for other events to depend on. Tasks with no parent dependencies (*i.e.:* root nodes in the DAG) can be inserted by setting both of the *num_parents* parameters to 0. If not set to NULL, the *free_op_data* callback will be invoked with the task's *op_data* pointer when all the outstanding references to the new task are released (with AXEfinish*).

Entries in the parent arrays can be IDs for tasks that have not been created yet. In this case those parent tasks will be considered incomplete until they are created and execute.

Note that this feature allows cycles to be created that can never be executed - make sure to avoid this. For example, never create a child of a barrier task before creating the barrier task. Also note that this function will not fail if you pass an "invalid" ID as a parent, so if while debugging you are having problems occur later on you may want to check the parents passed to this function.

The task ID is provided by the application in the task parameter. This can be generated by the application or by AXEgenerate_task_id. Applications must use caution when mixing both methods for ID creation to make sure the same ID is not used twice. In this case, it may be prudent to define a range for automatic ID generation using AXEengine_attr_id_range and manually generate IDs outside that range.

The value in *task_id* must not be reused, even if this function fails.

b) AXEcreate_barrier_task(AXE_engine_t engine, AXE_task_t task_id,

AXE_task_op_t op, void *op_data, AXE_task_free_op_data_t free_op_data)

This function adds a task to an engine that creates a necessary parent dependency on all the current leaf nodes of the engine's DAG. In other words, this task serves as a synchronization point for all the current tasks in the engine's DAG. For the purposes of this function a "leaf node" is a node with no necessary children (it may have sufficient children).

If the *op* parameter is NULL, no operation will be invoked for this task and this task is solely a placeholder in the graph for other tasks to depend on. If not set to NULL, the *free_op_data* callback will be invoked on the task's *op_data* pointer when all the outstanding references to the new task are released (with AXEfinish*).

The task ID is provided by the application in the task parameter. This can be generated by the application or by AXEgenerate_task_id. Applications must use caution when mixing both methods for ID creation to make sure the same ID is not used twice. In this case, it may be prudent to define a range for automatic ID generation using AXEengine_attr_id_range and manually generate IDs outside that range.

The value in *task_id* must not be reused, even if this function fails.

c) **AXEremove(AXE_engine_t engine, AXE_task_t task_id, /* OUT */ AXE_remove_status_t *remove_status)**

This function attempts to removes a task from an engine. Tasks that have child dependencies may not be removed (*i.e.:* only leaf nodes may be removed from the graph).

The result of the attempt is returned in **remove_status*. If the task has not started running, the task will be canceled and **remove_status* will be set to AXE_CANCELED. If the task's *op* function is in the middle of execution, the task will not be canceled and **remove_status* will be set to AXE_NOT_CANCELED. If the task has already finished, the task will not be canceled and **remove_status* will be set to AXE_ALL_DONE. If the task has any necessary or sufficient children this function will return AXE_FAIL and **remove_status* will not be set.

This function does not release the task handle. AXEfinish* must still be called on *task_id* when the application is done with it.

d) **AE2remove_all(AE2_engine_t engine, /* OUT */ AXE_remove_status_t *remove_status)**

This function attempts to removes all tasks from an engine. The result of the attempt is returned in **remove_status*. If at least one task was canceled and all others (if any) were already complete, **remove_status* will be set to AXE_CANCELED. If at least one task's *op* function is in the middle of execution, the executing tasks will not be canceled and **remove_status* will be set to AXE_NOT_CANCELED. If all the tasks have already finished, **remove_status* will be set to AXE_ALL_DONE

This function does not release the task handles. AXEfinish* must still be called on the task IDs when the application is done with them.

APIs for managing task information:

a) **AXEget_op_data(AXE_engine_t engine, AXE_event_t task_id, /* OUT */ void **op_data)**

This function retrieves the *op_data* pointer for a task, returning it in **op_data*. This function may be used to retrieve the *op_data* pointer for parent tasks, when a tasks operation is executed.

b) **AXEget_status(AXE_engine_t engine, AXE_event_t task_id, /* OUT */ AXE_status_t *status)**

This function retrieves the status of a task, and can be used to determine if a task has completed. Note that if the task has not been created, this function will succeed and **status* will be set to AXE_TASK_NOT_INSERTED.

APIs for managing completion of tasks:

a) **AXEwait(AXE_engine_t engine, AXE_task_t task_id)**

This function blocks the program execution until a task completes execution. If the task depends on other tasks in the engine, then those parent tasks will also have completed when this function returns. If the task had completed before this function is called, this function will immediately return success. If the task is canceled while waiting or was canceled before this function is called, this function will return failure.

Note that if the task has not been created, this function will wait until the task is created and runs to completion or is canceled.

b) **AXEfinish(AXE_engine_t engine, AXE_task_t task_id)**

This function tells the engine that the application is finished with this task handle. All engine-side resources will be released and the task's *free_op_data* callback will be invoked, if all the references to *task_id* have been released.

c) **AXEfinish_all(AXE_engine_t engine, size_t num_tasks, AXE_task_t task_ids[])**

This function tells the engine that the application is finished with a set of tasks. All engine-side resources for each task will be released and each task's *free_op_data* callback will be invoked, if all the references to a task have been released.

6. Client Use of the Asynchronous Execution Engine: Caveats

Some tricky programming issues are worth mentioning here regarding the usage of this asynchronous execution engine. First of all, a programmer should not assume the execution order between two tasks' operations have a dependency relationship outside that specified by the DAG. In other words, a program may have errors if the program logic assumes one asynchronous operation happens before another one, if no dependency relationship is explicitly established between the two tasks through the API routines above. In addition, two asynchronous operations that have no dependency between them may execute concurrently by helper threads, therefore programmers must ensure asynchronous operations are thread-safe (i.e. protecting concurrently accessed shared data with mutexes, etc). Furthermore, the asynchronous operations offloaded to this execution engine should not have any data-racing issues with the operations (like computation work) that execute on the main program thread.

7. Implementation of the Engine (Open to Discussion)

7.1 Components

<We should add a description of the internal functionality of AXE as implemented here>

7.2 Main Success Story for Adding a Task to the Engine and Executing it

A. Select a function that will be performed and set up input and output data parameters in the op_data pointer.

B. Determine the parent tasks.

C. Create the task via the API call.

D. Wait for the task to finish. Status can be checked via an API call and will be set to DONE when the task is complete.

E. When the task is complete, read the task's output data.

F. At this point, the task can be destroyed by the user. The task's memory was allocated and set up by the client, so the library does nothing with the op_data pointer. If the task is still needed internally by the engine for some purpose, its final deletion time will be handled by the library.

8. Misc. Design Issues

8.1 Reference Counting and Cleanup

Tasks in the engine are reference counted, with an optional user-defined 'free' function invoked on the op_data when the ref count drops to zero. A task's ref count begins at one when it is created, is incremented when it is used as a parent dependency, and is decremented when 'finish' is called on one of its handles. Note that the ref count on all 'sufficient' parents of a task is incremented when a child task is created, and the engine is responsible for decrementing the ref count on all parents when a child task's operation is invoked and the parent task is no longer needed by the child.

8.2 Fairness

The algorithm is fair: all tasks are enqueued in the order that they are received and attempts at dequeueing are made in FIFO order. Stalled tasks can only significantly impede their children.

8.3 Thread Availability

AXE makes the guarantee that, if an engine is created with *n* threads, then all *n* of those threads will be available for simultaneous execution. At no time will a thread in the thread pool enter a waiting state while there is a task available for execution. This is required to support algorithms that may need a certain number of tasks to execute simultaneously in order to make progress.

8.4 Concurrent Access to Internal Data Structures

The only concurrent threads that run inside the engine are those in the thread pool, modifying data structures when tasks complete. There is no "main engine" thread – new tasks are added to the engine's data structures by the application thread that calls the task creation call. If a thread in the pool is idle when a task is created, the engine activates that thread and gives it a task to operate on. When a thread concludes execution of a task's operation, it performs housekeeping for that task (including moving child tasks from the 'waiting for a parent' state to the 'scheduled' state, if possible) and then attempts to acquire another task to execute (it may also wake up other idle threads, if multiple tasks become schedulable). If no tasks are available for execution, the thread moves to an idle state.

8.5 Usage of OpenPA and pthreads Packages

<We should mention how AXE depends on and uses OPA and pthreads>

8.6 Core Affinity

<We should think about if core affinity for AXE threads would make sense, and how we would implement it>

9. Performance Metrics

<We should try to measure overhead for the engine, in various sensible ways, and get some numbers in here>

10.Download & Build

<Need instructions for configuring and building AXE>

Repository URL is: http://svn.hdfgroup.uiuc.edu/axe/trunk/

Depends on the OpenPA package, located here: http://trac.mpich.org/projects/openpa

Revision History

January 17, 2013:	Version 1 sent to Quincey Koziol and Mohamad Chaarawi for comment
January 22, 2013:	Version 2 incorporated comments from Quincey Koziol and Mohamad Chaarawi
January 23, 2013:	Version 3 incorporated comments on the version 2 from Quincey Koziol
January 24, 2013:	Version 3.1 incorporated tweaks from Quincey Koziol
January 25, 2013:	Version 3.2 incorporated tweaks from Chao Mei
January 28, 2013:	Version 4, large revisions to make engine more powerful and simplify some aspects of the interface, Quincey Koziol
February 10, 2013:	Version 5, major design additions and some API changes. Edits by Dana Robinson.

February 12, 2013:	Version 6, further design and API refinements. Edits by Quincey Koziol.
February 12, 2013:	Version 7, spelling, grammar and cohesiveness pass. Edits by Quincey Koziol.
February 12, 2013:	Version 8, minor correction to the task status struct. Edits by Quincey Koziol.
May 15, 2013:	Version 8-ra, spelling, grammar and cohesiveness pass. Edits by Ruth Aydt.
June 29, 2014:	Version 9, strong edit pass. Edits by Quincey Koziol.
June 30, 2014:	Version 10, minor edit pass before initial publication, added repository locations for AXE and OpenPA. Edits by Mohamad Chaarawi, Quincey Koziol.
June 30, 2014:	Version 11, remove out-of-date implementation notes, update design issues. Edits by Neil Fortner, Quincey Koziol.