Composable Asynchronous Communication Graphs and Streams in MPI

Quincey Koziol, Amazon, qkoziol@amazon.com
Ryan Grant, Queens' University, ryan.grant@queensu.ca

The MPI standard has historically employed a "weak" progress model for overalapping communication, I/O, and computation with 'nonblocking' API calls. Weak progress means that the MPI library must be entered for nonblocking operations to be guaranteed of making progress toward completion. A "strong" progress model would mean that nonblocking operations would progress asynchronously toward completion, without requiring that the MPI library be entered for that progress to occur.

This proposal describes extensions to MPI that allow applications to request strong progress from an MPI implementation, and to optionally perform 'true' asynchronous operations.

Comment [QK1]: Is there a definitive description of weak and strong progress to refer to?

Add 'strong' progress to MPI Init, or session init

Revision History

Version Number	Date	Comments
v1-9	Mar-Sept, 2022	Hand-written lab notebook circulated informally amongst collaborators
v10	Sept. 14, 2022	Shared with MPI Collectives/Persistence Working Group
v11	Sept. 30, 2022	Shared at the MPI Forum.
v12-14	Oct. 17, 2022	Shared with HPE and NVIDIA.
v15	Oct. 18, 2022	Weekly discussions w/Ryan Grant.

Contents

List of Figures 5

List of Listings 6

1. Introduction 7

2. Background: MPI Today 7

3. Motivation / Etc. 8

4. Overview 8
- 4.1. Deferred Operations . . . 8
  - 4.1.1. Data Dependencies For Deferred Operations . . . 8
- 4.2. Aggregating Deferred Operations: MPI Graphs and Streams . . . 9
  - 4.2.1. Errors When Executing Deferred Operations Asynchronously . . . 10
- 4.3. Strong Progress . . . 10

5. Approach 10

6. New API Routines 11
- 6.1. Deferred MPI Operations . . . 11
- 6.2. Graph Management . . . 12
  - 6.2.1. Graph Variables . . . 18
- 6.3. Stream Management . . . 21
- 6.4. Memory Operations . . . 23
- 6.5. Synchronization Operations . . . 23
- 6.6. Local MPI Data Types . . . 24

7. Use Cases 24
- 7.1. Use Case #1: Communication and Computation Overlap . . . 24
- 7.2. Use Case #2: Asynchronously Allocate and Receive Unknown Message . . . 26
- 7.3. Use Case #3: Prefetch compressed data from file and broadcast to other ranks . . . 27
- 7.4. Use Case #4: Offload Collective I/O For Writing Checkpoint File . . . 30
- 7.5. Use Case #5: Data Dependent Asynchronous Communication . . . 32

References 37

A. Appendix: An Abstract Data Movement Machine 37
- A.1. Data Containers . . . 37
- A.2. Container Operations . . . 37
  - A.2.1. Abstract Copy Operations . . . 37
  - A.2.2. Abstract Reduce Operations . . . 38
  - A.2.3. Abstract Comparison Operations . . . 38
  - A.2.4. Abstract Synchronization Operations . . . 38
  - A.2.5. Constants . . . 38
- A.3. Graphs . . . 39
  - A.3.1. Graph Task Execution Unit . . . 39
  - A.3.2. Graph Dependency Unit . . . 39
  - A.3.3. Graph Construction Rules . . . 44
  - A.3.4. Graph Variables . . . 44

A.4. Streams . . . 45
A.4.1. Stream Task Execution Unit . . . 46
A.4.2. Stream Construction Rules . . . 46
A.4.3. Stream Variables . . . 46
A.5. Scopes . . . 49

B. Appendix: Comparison to proposed Continuations and MPIX_Stream extensions 50

C. Appendix: What this proposal is *not* 51

D. Appendix: Details for Asynchronous Operations Option #2 52

List of Figures

1. Task dependencies . . . 40
2. General form of unconditional organizational graph dependencies . . . 40
3. Common forms of unconditional organizational graph dependencies . . . 41
4. Threshold organizational graph dependency . . . 42
5. Branch organizational graph dependency . . . 42
6. Control flow operation . . . 43
7. Example graph for control flow destinations . . . 43
8. Connecting multiple task dependencies . . . 44
9. Connecting multiple organizational dependencies . . . 45
10. Combining task and organizational dependencies . . . 46
11. Combining tasks, organizational dependencies, and control flow operations to create loop in graph . . . 47
12. Combining tasks, organizational dependencies, and sub-graphs . . . 47
13. Tasks in a stream . . . 48
14. Tasks and sub-graphs in a stream . . . 48

List of Listings

1. Deferred operations added to graphs 9
2. Deferred operations added to streams 10
3. Data-dependent communication operations 25
4. Asynchronously allocate space for and receive message of unknown length 27
5. Asynchronously prefetch a compressed file on rank 0 and broadcast the decompressed buffer to other ranks 30
6. Asynchronously write checkpoint file with collective I/O 32
7. Data Dependent Asynchronous Operations 35

1. Introduction

The goal of this RFC is to present a proposal for improving performance of MPI applications in 3 primary ways: reduce synchronization penalties for collective operations, fully overlap communication, I/O, and computation, and enable more MPI operations to execute concurrently with application computation.

Expand on the goals

2. Background: MPI Today

MPI currently provides two kinds of communication and I/O operations: blocking and nonblocking (which are also sometimes called "immediate"). Blocking operations do not return until the operation is complete (or an error occurs). Nonblocking operations return immediately and provide a "request" object (`MPI_Request`) to the caller, which can use the request to check for the operation's completion (or error). Initially, nonblocking operations appear to meet the goals for strong progress with asynchronous operation, but there are many restrictions and caveats to them, as shown below.

To begin with, a simple set of blocking MPI operations might look like this:

Rank 0	Rank 1

`MPI_Send(...)`
`MPI_Send(...)`
`...`
`MPI_Send(...)`
	`MPI_Recv(...)`
	`MPI_Recv(...)`
	`...`
	`MPI_Recv(...)`

Clearly, there's no way to overlap these communication operations with any computation, as all the MPI operations are blocking.

Attempting to use nonblocking MPI operations might look something like this:

Rank 0	Rank 1

`MPI_Isend(..., &req[0])`
`MPI_Isend(..., &req[1])`
`...`
`MPI_Isend(..., &req[n])`
`[Compute]`
`MPI_Waitall(..., n+1, req)`
	`MPI_Irecv(..., &req[0])`
	`MPI_Irecv(..., &req[1])`
	`...`
	`MPI_Irecv(..., &req[n])`
	`[Compute]`
	`MPI_Waitall(..., n+1, req)`

However, using nonblocking operations has many drawbacks:

a) Nonblocking operations are not guaranteed to make progress unless the MPI library is entered

b) An application must track individual nonblocking operations with a request object for each one

c) There is no way to indicate dependencies between nonblocking operations

d) There is no way to invoke a user operation for asynchronous execution[1]

e) Applications and middleware libraries can't easily nest asynchronous operations on or around the MPI API

[1] Generalized requests are not a solution that meets the requirements outlined here.

f) A result from one asynchronous operation can't be used as a "future" value for another operation

g) There are no asynchronous operations for memory allocation or release, as well as local memory movement

h) Many file operations are not supported

The asynchronous operations described in this document correct the drawbacks of nonblocking operations as well as put new capabilities into the hands of application developers.

3. Motivation / Etc.

Include benchmarks that show the benefit of overlapping communication and I/O with computation

4. Overview

Building a robust set of extensions to add "true" asynchronous operations to MPI can achieve the goal of improving application performance by enabling full overlap with computation, which can reduce the costs for communication and I/O to nearly zero.

Additional benefits of achieving the primary performance goals in an elegant and well-designed way are: an improved 'user experience' for developers using asynchronous (currently 'nonblocking') operations, hiding the latency of operations, exposing more opportunities for optimizing performance of MPI operations, enabling offload of more operations to networking hardware, and enabling applications to build powerful data movement orchestration operations.

4.1. Deferred Operations

This document describes two mechanisms for executing asynchronous MPI operations: defining an aggregated set of operations (a "graph") to execute later and executing operations immediately in an ordered manner (a "stream"). Each of those mechanisms relies on a new concept: "deferred" MPI operations. Deferred operations are very similar to persistent operations, but are designed to cover all kinds of MPI operations, unlike persistent operations, which focus on communication operations.[2] [3]

Executing deferred MPI operations in a "fully" asynchronous manner, which are guaranteed to complete without re-entering the MPI implementation, requires strong progress, described in section 4.3.

Listings 1 and 2 are pseudocode examples that shows the approach, with graphs and streams.

Implementing deferred operations in this way allows for a single API definition for each operation, which returns a `MPI_Token` object that can be added to a graph or enqueued on a stream.

4.1.1. Data Dependencies For Deferred Operations

Deferred operations may have data dependencies on values produced in earlier operations, as well as produce values that later operations may wish to consume. MPI operations have two kinds of data dependencies: strong dependencies, which involve *handles* of objects, and weak dependencies, which involve the *contents* of objects. For example, a strong dependency is created by the `MPI_File` file handle produced from a deferred call to `MPI_File_open` that is used in a later deferred call to `MPI_File_get_size`. A weak

[2] Deferred operations can cover `MPI_File_open`, `MPI_Comm_create`, etc. along with `MPI_Send` and related operations that are covered by the set of persistent operations.

[3] Nonblocking operations will not work, because they are allowed to start execution immediately and are not compatible with the approach described here.

```
// Info keys control graph execution behavior
MPIX_Graph_create(session, info, &graph);
...
// Define deferred operations and add to graph
MPIX_File_open_def(..., &token);
MPIX_Graph_add(graph, &token, <dependency info>);
...
MPIX_File_read_def(..., &token);
MPIX_Graph_add(graph, &token, <dependency info>);
...
MPIX_File_close_def(..., &token);
MPIX_Graph_add(graph, &token, <dependency info>);
...
MPIX_Bcast_def(..., &token);
MPIX_Graph_add(graph, &token, <dependency info>);
...
// Create deferred operation token for graph
MPIX_Graph_def(graph, info, &token);
...
//  Create an [inactive] request for the graph token
MPIX_Execute_init(token, info, MPIX_OFFLOAD_CUDA, &cuda_stream, &req);
...
// Start execution of graph
MPIX_Start(req);
...
<compute or other overlap w/graph execution>
...
// Conclude graph's operations
MPI_Wait(&req, &status);
```

Listing 1. – Deferred operations added to graphs

data dependency is demonstrated by the data in a buffer from a deferred call to `MPI_Recv` being used as the source buffer for a deferred call to `MPI_Bcast` that has a dependency on the call to `MPI_Recv`.

4.2. Aggregating Deferred Operations: MPI Graphs and Streams

Graphs and streams are the core objects that "hold" deferred operations. Deferred operations can be added to a graph, with optional dependencies on other asynchronous operations. Dependencies between deferred operations can describe a directed graph and deferred operations without dependencies can execute concurrently. Deferred operations added to a stream execute in FIFO order, without requiring explicit dependencies, but also without the possible concurrent execution that is possible with graphs.[4]

MPI graphs have an additional supporting object: variables. MPI graphs allow the composition of Turing-complete data movement kernels using graph variables to control the execution of if/else blocks and loops. Graph variables are also used to parameterize graphs, allowing for their re-use with new inputs.

[4]For full details on graph and stream construction rules, see sections A.3.3 and A.4.2 respectively.

```
// Info keys control stream execution behavior
MPIX_Stream_create(session, info, MPIX_OFFLOAD_CUDA, &cuda_stream, &stream);
...
// Pause stream from processing operations that are added
MPIX_Stream_pause(stream);
...
// Define deferred operations and enqueue into stream
MPIX_File_open_def(..., &token);
MPIX_Stream_enqueue(stream, &token);
...
MPIX_File_read_def(..., &token);
MPIX_Stream_enqueue(stream, &token);
...
MPIX_File_close_def(..., &token);
MPIX_Stream_enqueue(stream, &token);
...
MPIX_Bcast_def(..., &token);
MPIX_Stream_enqueue(stream, &token);
...
// Start processing all the enqueued operations
MPIX_Stream_resume(stream);
...
// Ensure that all operations currently on stream have finished
MPIX_Stream_sync(stream);  // Optional
...
```

Listing 2. – Deferred operations added to streams

4.2.1. Errors When Executing Deferred Operations Asynchronously

There can be many asynchronous operations in an aggregation object[5], possibly executing concurrently. Additionally, data dependencies on information sent from other MPI ranks at runtime are possible. This complex and unpredictable environment argues against investing in mechanisms to resume/restart from failures.

Therefore, if an error occurs when executing a deferred operation in an aggregation object, the aggregation object will permanently stop scheduling new operations for execution. Existing operations that are executing will be allowed to complete, but no further operations will be started.

When an error occurs, only three actions on the aggregation object are possible: use the "introspection" API routines on the object (mainly to query about the failed operation, although any introspection operation can be called), duplicate the object (for possible possible "restart from the beginning"), and free the object[6].

4.3. Strong Progress

Describe enabling strong progress in MPI sessions, similar to enabling threading

5. Approach

Add de-

[5] An MPI graph or stream

[6] *You can autopsy or clone it before you bury it, but it's dead.*

6. New API Routines

6.1. Deferred MPI Operations

TODO: Describe memory operation APIs...

Discuss deferred operations, how they are similar / different to persistent and non-blocking operations.

Mention that deferred persistent, nonblocking, and partitioned operations are possible.

Create table of important deferred operations, including file I/O, and alloc / free mem.

MPIX_ZZZ_DEF(..., token)

-	...	existing parameters for operation
OUT	token	deferred operation token (handle)

MPIX_ZZZ_DEF corresponds to a deferred version of the `MPI_ZZZ` operation. The `token` object may be added to a graph or stream.

MPIX_OP_DEF(op, extra_state, token)

IN	op	function pointer to user callback to invoke
IN	extra_state	pointer to extra state for user callback
OUT	token	deferred operation token (handle)

MPIX_OP_DEF creates a deferred invocation of a user callback.[7] User callbacks invoked through this mechanism are designed to be "support" routines, not intensive computations. The `token` object may be added to a graph or stream.

NOTE: The definition of the user callback would probably look like:

```
void (*op)(MPIX_Graph *graph, void *extra_state)
```

Possibly other parameters?

The `op` callback receives a pointer to the `MPIX_Graph` that it was invoked from, as well as the `extra_state` specified in the MPIX_OP_DEF call. The `graph` parameter may be used for calls to MPIX_GRAPH_VAR_GET and MPIX_GRAPH_VAR_SET, to access the graph's variables.

NOTE: This is designed as an "escape hatch" for user operations that should execute in a graph or stream, but aren't covered by the current deferred operations in the MPI API.

There are potentially lots of sharp edges here, so we should add more cautionary notes and user guidance.

MPIX_EXECUTE(token, info, loc, loc_info)

IN	token	deferred operation token (handle)
IN	info	info object (handle)
IN	loc_type	offload type (handle)
IN	loc_info	pointer to offload location info (choice)

MPIX_EXECUTE executes the deferred operation specified in `token`. The deferred token object is <u>not</u> consumed by this operation, allowing it to be used for multiple execute operation. Hints may be provided by the `info` argument. The execution offload type of the operation is specified by `loc_type`, with additional information provided in `loc_info`. Valid values for `loc_type` are given in <side document>, but implementations must at least support MPIX_LOC_INPLACE, for local execution without offload.

CUDA, ROCM,

[7] MPI generalized requests are similar, but are not able to be invoked from a graph or stream.

MPIX_EXECUTE_INIT(token, info, loc_type, loc_info, req)

IN	token	deferred operation token (handle)
IN	info	info object (handle)
IN	loc_type	offload type (handle)
IN	loc_info	pointer to offload location info (choice)
OUT	req	request object (handle)

MPIX_EXECUTE_INIT is a persistent version of MPIX_EXECUTE, returning an inactive request object in `req`. The request must be passed to a function in the MPI_Start family of operations to become active.

MPIX_TOKEN_FREE(token)

INOUT	token	deferred operation token (handle)

MPIX_TOKEN_FREE releases a deferred operation and sets the value of `token` to `MPIX_TOKEN_NULL`. Freeing a token has no effect on deferred operations or requests which have used the token already.

6.2. Graph Management

TODO: Describe graph management APIs...

Probably need to introduce operation IDs

MPIX_GRAPH_CREATE(session, info, graph)

IN	session	session (handle)
IN	info	info object (handle)
OUT	graph	graph object (handle)

MPIX_GRAPH_CREATE creates a new graph object in the specified session. An info object is provided in order to support optimization hints and other information that may be nonstandard.

MPIX_GRAPH_DUP(graph, info, newgraph)

IN	graph	graph (handle)
IN	info	info object (handle)
OUT	newgraph	copy of graph (handle)

MPIX_GRAPH_DUP duplicates the existing graph `graph`. Hints provided by the argument `info` are associated with the output graph `newgraph`.

....

MPIX_GRAPH_ADD(graph, token, op_id, dep_op_id)

INOUT	graph	graph (handle)
INOUT	token	deferred operation token (handle)
OUT	op_id	operation ID (non-negative integer)
IN	dep_op_id	operation ID (non-negative integer)

MPIX_GRAPH_ADD adds the deferred operation token to the graph graph. The graph takes ownership of the deferred token object, freeing it and setting token to MPIX_TOKEN_NULL.

Optionally an operation ID op_id can be returned to the application to refer to this operation in the graph, to create dependencies on it. NULL may be passed to indicate that no operation ID should be returned.

dep_op_id may also optionally be provided, to create a dependency on an earlier operation already added to the graph. MPIX_DEP_NONE may be passed to indicate that the added operation has no parent dependency.

NOTE: Operation IDs are only valid for creating dependencies between operations in the same graph. Using an operation ID from one graph in another graph is not supported and may cause undefined behavior.

....

MPIX_GRAPH_JOIN(graph, count, array_of_dep_op_ids, op_id)

INOUT	graph	graph (handle)
IN	count	array size (non-negative integer)
IN	array_of_dep_op_ids	array of dependent operation IDs (array of non-negative integers)
OUT	op_id	operation ID (non-negative integer)

MPIX_GRAPH_JOIN provides the capability to have many-parent to one-child relationships in a graph. MPIX_GRAPH_JOIN produces an operation ID op_id that can be used as a dependent operation ID for further operations in the graph that depend on all the operations in array_of_dep_op_ids completing before the operation can execute. All of the operation IDs in array_of_dep_op_ids must be from operations in graph.

. . . .

```
MPIX_GRAPH_IF_ELSE(graph, var, true_graph, false_graph,
                   op_id, dep_op_id)
```

INOUT	graph	graph (handle)
IN	var_id	variable identifier (non-negative integer)
INOUT	true_graph	graph (handle)
INOUT	false_graph	graph (handle)
OUT	op_id	operation ID (non-negative integer)
IN	dep_op_id	operation ID (non-negative integer)

MPIX_GRAPH_IF_ELSE adds a conditional dependency to `graph`. If the value of the graph variable `var_id` is non-zero, the `true_graph` graph will be the child of the MPIX_GRAPH_IF_ELSE dependency and if `var` is zero, the `false_graph` graph will be the child of the dependency.

The MPIX_GRAPH_IF_ELSE operation *copies* both the `true_graph` and the `false_graph`, allowing them to be re-used for other purposes (and eventually released with MPIX_GRAPH_FREE). `MPIX_GRAPH_NULL` may be passed for either the true or false graph handles, indicating that there is no child dependency for that condition.

Comment [QK2]: Or increments the refcount on?

Optionally an operation ID `op_id` can be returned to the application to refer to this operation in the graph, to create dependencies on it. `dep_op_id` may also optionally be provided, to create a dependency on an earlier operation already added to the graph.

NOTE: Examining the graph variable within the MPIX_GRAPH_IF_ELSE operation could have weak data dependencies on other concurrently executing operations in the graph and create race conditions when the graph is executed. Application developers can use other organizational dependencies as preludes to MPIX_GRAPH_IF_ELSE to eliminate such race conditions.

. . . .

`MPIX_GRAPH_WHILE(graph, var_id, sub_graph, op_id, dep_op_id)`

INOUT	graph	graph (handle)
IN	var_id	graph variable (handle)
INOUT	sub_graph	graph (handle)
OUT	op_id	operation ID (non-negative integer)
IN	dep_op_id	operation ID (non-negative integer)

MPIX_GRAPH_WHILE adds a loop to `graph`. If the value of the graph variable `var_id` is non-zero, the `sub_graph` graph will be executed.

The MPIX_GRAPH_WHILE operation *copies* the `sub_graph`, allowing it to be re-used for other purposes (and eventually released with MPIX_GRAPH_FREE).

Comment [QK3]: Or increments the refcount on?

Optionally an operation ID `op_id` can be returned to the application to refer to this operation in the graph, to create dependencies on it. `dep_op_id` may also optionally be provided, to create a dependency on an earlier operation already added to the graph.

NOTE: Examining the graph variable within the MPIX_GRAPH_WHILE operation could have weak data dependencies on other concurrently executing operations in the graph and create race conditions when the graph is executed. Application developers can use other organizational dependencies as preludes to MPIX_GRAPH_WHILE to eliminate such race conditions.

. . . .

`MPIX_GRAPH_GOTO(graph, dest_op_id, dep_op_id)`

INOUT	graph	graph (handle)
IN	dest_op_id	operation ID (non-negative integer)
IN	dep_op_id	operation ID (non-negative integer)

MPIX_GRAPH_GOTO transfers the flow of control within `graph` to another operation in the graph, `dest_op_id`.

Optionally an operation ID, `dep_op_id`, may be provided, to create a dependency on an earlier operation already added to the graph.

NOTE: It is possible to create > 1 MPIX_GRAPH_GOTO operation as a dependency on `dep_op_id`, starting concurrent execution of the `dest_op_id` for each MPIX_GRAPH_GOTO operation.

. . . .

`MPIX_GRAPH_SET_DEP_OP(graph, op_id, dep_op_id)`

INOUT	graph	graph (handle)
IN	op_id	operation ID (non-negative integer)
IN	dep_op_id	operation ID (non-negative integer)

MPIX_GRAPH_SET_DEP_OP sets or modifies the `dest_op_id` for `op_id` in `graph`. The value 0 may be passed for `dest_op_id` to remove the dependent operation for `op_id`. `dep_op_id` may not be set to the same value as `op_id`.

NOTE: Application developers should take care not to create dependency loops.

. . . .

`MPIX_GRAPH_NOP(graph, op_id, dep_op_id)`

INOUT	graph	graph (handle)
IN	op_id	operation ID (non-negative integer)
IN	dep_op_id	operation ID (non-negative integer)

MPIX_GRAPH_NOP creates a "no op" operation in `graph`.

Optionally an operation ID `op_id` can be returned to the application to refer to this operation in the graph, to create dependencies on it. `dep_op_id` may also optionally be provided, to create a dependency on an earlier operation already added to the graph.

NOTE: MPIX_GRAPH_NOP and MPIX_GRAPH_SET_DEP_OP may be useful when creating and updating 'placeholder' operations as destinations when constructing complex control flows for graphs.

. . . .

`MPIX_GRAPH_SET_PARAM(graph, var_id, value / alias_var_id)`

INOUT	graph	graph (handle)
IN	var_id	parameter variable identifier (non-negative integer)
IN	value or alias_var_id	pointer to parameter value (choice) or alias variable identifier (non-negative integer)

MPIX_GRAPH_SET_PARAM is overloaded to accept either a variable ID or a pointer to a value. If a variable ID is passed, the graph parameter variable, `var_id`, must be a reference parameter variable and if a pointer to a value is passed, the graph parameter variable must be a value parameter (see MPIX_GRAPH_VAR_CREATE for more information). Graph parameter variables can be set or changed at any time before or after MPIX_GRAPH_DEF is called.

. . . .

`MPIX_GRAPH_DEF(graph, info, token)`

INOUT	graph	graph (handle)
IN	info	info object (handle)
IN	token	deferred operation token (handle)

MPIX_GRAPH_DEF binds the parameters for the graph `graph` to a deferred operation token `token`. Further changes to the parameters for the graph do not change the parameters for the initialized graph associated with `token`. The deferred operation token representing the initialized graph may be added to another graph (with MPIX_GRAPH_ADD), enqueued in a stream (with MPIX_STREAM_ENQUEUE), or executed with MPIX_EXECUTE or MPIX_IEXECUTE.

NOTE: If any parameter variables are defined but not set for the graph (with MPIX_GRAPH_SET_PARAM), MPIX_GRAPH_DEF will fail.

. . . .

`MPIX_GRAPH_FREE(graph)`

INOUT	graph	graph (handle)

MPIX_GRAPH_FREE frees `graph` and sets it to `MPIX_GRAPH_NULL`.

6.2.1. Graph Variables

TODO: Describe graph variables and APIs...

MPIX_GRAPH_VAR_CREATE(graph, var_id, scope, type)

INOUT	graph	graph object (handle)
IN	var_id	variable identifier (non-negative integer)
IN	scope	scope of variable (handle)
IN	datatype	datatype of variable (handle)

MPIX_GRAPH_VAR_CREATE creates a new variable in the specified graph, graph, identified with var_id. datatype must be an atomic MPI datatype[8]. The scope of the variable is one of the following values, with the associated meaning:

Maybe a better word than 'scope'?

- **MPIX_GRAPH_VAR_VAL_PARAM** – a variable that is used as a parameter for the graph, and must initialized with a value using MPIX_GRAPH_SET_PARAM before the graph starts execution.
- **MPIX_GRAPH_VAR_REF_PARAM** – a variable that is an alias for a variable in the application or another graph variable in an outer scope, and must initialized with MPIX_GRAPH_SET_PARAM before the graph starts execution.
- **MPIX_GRAPH_VAR_LOCAL** – a variable that is initialized to "all zeroes" when the graph starts execution, and is not aliased to a variable in an outer scope.

Variables may be declared at any point during a graph's definition, but assignments for all value or reference parameter variables defined anywhere in the graph must be provided before MPIX_GRAPH_DEF is called on the graph.

Reference parameters are aliases for application variables or graph variables in an outer graph scope. The reference parameter can be thought of an an RMA window on the aliased application or outer graph variable. The value in the aliased variable is not valid for inspection through the aliased variable until the graph completes execution.

Graph variables are valid sources and destinations where memory buffers are specified for **all** deferred MPI data movement operations, such as MPIX_REDUCE_DEF and MPIX_SEND_DEF, etc. Graph variables can also be used for 'IN' parameters to deferred MPI operations and as 'future' values from deferred MPI operations with 'OUT' parameters, such as MPI_FILE_OPEN, etc.

Graph variables may be of a "local" MPI type, allowing a variable to be used as a parameter of the same type for deferred operations. For example, a graph variable may be created with a type of MPI_TYPE_FILE and used in all the places where an MPI_FILE_* operation accepts (an an 'IN' parameter) or returns (as an 'OUT' parameter) a parameter of type MPI_File. Local MPI data types are described in section 6.6.

Should an application wish to set the value of a graph variable during execution of a graph or preserve a variable's value before the graph completes execution, the MPIX_COPY_MEM_DEF operation should be used with a graph variable as the source or destination buffer (or both).

[8] Including MPI_AINT, MPI_COUNT, and `local' MPI data types like MPI_FILE, MPI_STATUS, etc.

MPIX_GRAPH_VAR_SET(graph, var_id, value)

INOUT	graph	graph (handle)
IN	var_id	parameter variable identifier (non-negative integer)
IN	value	pointer to new value for variable (choice)

MPIX_GRAPH_VAR_SET sets the value of variable var_id in graph graph to the value pointed to by the value pointer.

Graph variables set before MPIX_GRAPH_DEF is called will be associated with the graph token created by MPIX_GRAPH_DEF and later calls to MPIX_GRAPH_VAR_SET will not affect the values for the variables associated with that token.

Graph variables set to values in a user callback (with MPIX_OP_DEF) are reflected in the values for those variables when the callback completes. Race conditions around graph variables may exist depending on the graph's operations and an application developer should use operation dependencies within the graph to avoid them.

MPIX_GRAPH_VAR_GET(graph, var_id, value)

INOUT	graph	graph (handle)
IN	var_id	parameter variable identifier (non-negative integer)
OUT	value	pointer to a location to place the value for variable (choice)

MPIX_GRAPH_VAR_GET retrieves the value of variable var_id in graph graph into the location pointed to by the value pointer.

Graph variable values retrieved within a user callback (with MPIX_OP_DEF) reflect the value of those variables when MPIX_GRAPH_VAR_GET is invoked. Race conditions around graph variables may exist depending on the graph's operations and an application developer should use operation dependencies within the graph to avoid them.

. . . .

```
MPIX_GRAPH_VAR_REDUCE(graph, in_var1_id, in_var2_id,
                      var_op, out_var_id, op_id, dep_op_id)
```

INOUT	graph	graph (handle)
IN	in_var1_id	variable identifier (non-negative integer)
IN	in_var2_id	variable identifier (non-negative integer)
IN	var_op	operation (handle)
IN	out_var_id	variable identifier (non-negative integer)
IN	op_id	operation ID (non-negative integer)
IN	dep_op_id	operation ID (non-negative integer)

MPIX_GRAPH_VAR_REDUCE is similar to MPI_REDUCE_LOCAL, but performs an operation `op` on two graph variables, `in_var1_id` and `in_var2_id` and places the result in `out_var_id`. All three variables must belong to the same graph, `graph`.

`var_op` may be any of the predefined reduction operations defined for MPI_REDUCE, with the exception of MPI_MAXLOC and MPI_MINLOC. Additionally, the following operations are defined for MPIX_GRAPH_VAR_REDUCE:

- **MPIX_LESS_THAN_OR_EQUAL** – produces a non-zero or TRUE value when the value of `in_var1_id` is less than or equal to the value of `in_var2_id`, i.e. $in\_var1\_id \leq in\_var2\_id$.
- **MPIX_LESS_THAN** – produces a non-zero or TRUE value when the value of `in_var1_id` is less than the value of `in_var2_id`, i.e. $in\_var1\_id < in\_var2\_id$.
- **MPIX_EQUAL** – produces a non-zero or TRUE value when the value of `in_var1_id` is equal to the value of `in_var2_id`, i.e. $in\_var1\_id = in\_var2\_id$.
- **MPIX_NOT_EQUAL** – produces a non-zero or TRUE value when the value of `in_var1_id` is not equal to the value of `in_var2_id`, i.e. $in\_var1\_id \neq in\_var2\_id$.
- **MPIX_GREATER_THAN** – produces a non-zero or TRUE value when the value of `in_var1_id` is greater than the value of `in_var2_id`, i.e. $in\_var1\_id > in\_var2\_id$.
- **MPIX_GREATER_THAN_OR_EQUAL** – produces a non-zero or TRUE value when the value of `in_var1_id` is greater than or equal to the value of `in_var2_id`, i.e. $in\_var1\_id \geq in\_var2\_id$.

Optionally an operation ID `op_id` can be returned to the application to refer to this operation in the graph, to create dependencies on it. `dep_op_id` may also optionally be provided, to create a dependency on an earlier operation already added to the graph.

6.3. Stream Management

TODO: Describe stream management APIs...

`MPIX_STREAM_CREATE(session, info, loc_type, loc_info, stream)`

IN	session	session (handle)
IN	info	info object (handle)
IN	loc_type	offload type (handle)
IN	loc_info	pointer to offload info (choice)
OUT	stream	stream object (handle)

MPIX_STREAM_CREATE creates a new stream object in the specified session. An info object is provided in order to support optimization hints and other information that may be nonstandard. The execution offload type of the operation is specified by `loc_type`, with additional information provided in `loc_info`. Valid values for `loc_type` are given in <side document>, but implementations must at least support MPIX_LOC_INPLACE, for local execution without offload.

CUDA, ROCM, pthread, etc.

. . . .

`MPIX_STREAM_ENQUEUE(stream, token)`

INOUT	stream	stream (handle)
INOUT	token	deferred operation token (handle)

MPIX_STREAM_ENQUEUE adds the deferred operation `token` to the stream `stream`. The stream takes ownership of the deferred token object, freeing it and setting `token` to `MPIX_TOKEN_NULL`.

Deferred operations enqueued in a stream are executed in the order they are added to the stream, i.e. 'FIFO' ordering. Deferred operation tokens may be individual operations, e.g. from MPIX_SEND_DEF, MPIX_FILE_OPEN_DEF, etc, or entire graphs, i.e. tokens returned from MPIX_GRAPH_DEF.

. . . .

`MPIX_STREAM_PAUSE(stream)`

INOUT	stream	stream (handle)

MPIX_STREAM_PAUSE pauses execution of deferred operations on a stream, `stream`. Deferred operations may continue to be enqueued on the stream, but none will begin executing until MPIX_STREAM_RESUME is called.

. . . .

`MPIX_STREAM_IS_PAUSED(stream, paused)`

IN	stream	stream (handle)
OUT	paused	flag indicating stream is paused (logical)

MPIX_STREAM_IS_PAUSED queries whether execution of deferred operations on a stream, `stream`, is paused.

. . . .

`MPIX_STREAM_RESUME(stream)`

INOUT	stream	stream (handle)

MPIX_STREAM_RESUME resumes execution of deferred operations on a stream, `stream`.

. . . .

`MPIX_STREAM_SYNC(stream)`

INOUT	stream	stream (handle)

MPIX_STREAM_SYNC blocks until all enqueued operations on `stream` at the time it is called have finished execution. Deferred operations enqueued (with another thread) after this routine is called, but before it returns, are not included in the list of operations blocked for.

NOTE: If MPIX_STREAM_SYNC is called on a paused stream, the stream will resume and complete the deferred operations enqueued when MPIX_STREAM_SYNC was called.

. . . .

`MPIX_STREAM_ISYNC(stream, request)`

INOUT	stream	stream (handle)
OUT	request	request object (handle)

MPIX_STREAM_ISYNC is a nonblocking form of MPIX_STREAM_SYNC. The request, `request`, will complete when all the enqueued operations on `stream` at the time MPIX_STREAM_ISYNC is called have finished execution. Deferred operations enqueued after this routine is called are not included in the list of operations the request applies to.

NOTE: If MPIX_STREAM_ISYNC is called on a paused stream, the stream's execution is **not** resumed. A paused stream must be resumed with either MPIX_STREAM_RESUME or MPIX_STREAM_SYNC.

. . . .

`MPIX_STREAM_FREE(stream)`

INOUT	stream	stream (handle)

MPIX_STREAM_FREE frees `stream` and sets it to MPIX_STREAM_NULL.

6.4. Memory Operations

TODO: Describe memory operation APIs...

`MPIX_COPY_MEM(dst, src, size)`

INOUT	dst	pointer to beginning of destination memory segment (choice)
IN	src	pointer to beginning of sourc memory segment (choice)
IN	size	size of memory segment in bytes (non-negative integer)

MPIX_COPY_MEM copies `size` bytes from memory area `src` to memory area `dst`. If `dst` and `src` overlap, behavior is undefined.

Add deferred version of this routine also

NOTE: This is the same behavior as Standard C `memcpy`.

Maybe we want memmove also / instead?

NOTE: MPIX_COPY_MEM_DEF is the mechanism for setting / getting values for graph variables.

6.5. Synchronization Operations

TODO: Describe new synchronization operation APIs...

`MPIX_REMOTE_BARRIER(comm, watched, info)`

IN	comm	communicator (handle)
IN	watched	rank of watched MPI process (integer)
IN	info	info object (handle)

MPIX_REMOTE_BARRIER blocks the caller until the `watched` MPI process has called it. For the `watched` process, MPIX_REMOTE_BARRIER is local and never blocks. An info object is provided in order to support optimization hints and other information that may be nonstandard.

Add deferred version of this routine also

NOTE: MPIX_REMOTE_BARRIER_DEF can be used to create inter-graph dependencies between concurrently executing graphs.

C Typedefs	Local Type Name
MPI_Comm	MPI_LOCAL_TYPE_COMM
MPI_Datatype	MPI_LOCAL_TYPE_DATATYPE
MPI_Errhandler	MPI_LOCAL_TYPE_ERRHANDLER
MPI_File	MPI_LOCAL_TYPE_FILE
MPI_Group	MPI_LOCAL_TYPE_GROUP
MPI_Info	MPI_LOCAL_TYPE_INFO
MPI_Message	MPI_LOCAL_TYPE_MESSAGE
MPI_Op	MPI_LOCAL_TYPE_OP
MPI_Request	MPI_LOCAL_TYPE_REQUEST
MPI_Session	MPI_LOCAL_TYPE_SESSION
MPI_Status	MPI_LOCAL_TYPE_STATUS
const char *	MPI_LOCAL_TYPE_STRING
MPI_Win	MPI_LOCAL_TYPE_WIN

Table 1 – C Typedefs and equivalent Local Data Types

6.6. Local MPI Data Types

TODO: Describe local MPI data types

TODO: Note 'const char *' as MPI_LOCAL_TYPE_STRING

7. Use Cases

Use cases that demonstrate the need and applicability of these new capabilities are given below.

7.1. Use Case #1: Communication and Computation Overlap

This example code demonstrates overlapping compute with communication, including: broadcasting the direction of neighbors to exchange with for the next timestep computation and a exchange of data between neighbors, all of which occurs asynchronously, overlapping the current compute step.

```
MPIX_Graph graph;
MPIX_Token token;
MPI_Request req;
enum {BCAST_VAL, DECR_VAL}; // Graph variable IDs

// Create graph for each timestep
MPIX_Graph_create(session, info, &graph);

// Create a value parameter graph variable to hold the broadcast information
MPIX_Graph_var_create(&graph, BCAST_VAL, MPIX_GRAPH_VAR_VAL_PARAM, MPI_INT);

// Create a local graph variable to hold the constant '-1'
```

```c
int dec_value = -1;
MPIX_Graph_var_create(&graph, DECR_VAL, MPI_INT);
MPIX_Graph_var_set(&graph, DECR_VAL, &dec_value);

// Define deferred bcast w/a graph variable and add to graph
// Note: The graph variable ID is passed for the buffer pointer
unsigned bcast_op_id = 0;
MPIX_Bcast_def(BCAST_VAL, ..., &token);
MPIX_Graph_add(&graph, &token, &bcast_op_id, MPIX_DEP_NONE);

// Define sub-graph for 'while' loop
MPIX_Graph_create(session, info, &while_graph);
// Exchange with neighbor
unsigned sendrecv_op_id = 0;
MPIX_Sendrecv_def(..., &token);
MPIX_Graph_add(&while_graph, &token, &sendrecv_op_id, MPIX_DEP_NONE);
// Decrement loop counter
// NOTE: Use of var IDs from outer graph is OK, as variable IDs are resolved
 ↪  when MPIX_Graph_def is called
MPIX_Graph_var_reduce(&while_graph, BCAST_VAL, DECR_VAL, MPI_SUM, BCAST_VAL,
 ↪  NULL, sendrecv_op_id);

// While value of var ID 'BCAST_VAL' is non-zero, execute sub-graph
MPIX_Graph_while(&graph, BCAST_VAL, &while_graph, NULL, bcast_op_id);

// Free sub-graph for while loop
MPIX_Graph_free(&while_graph);

while(<more timesteps>) {
  // Set parameter (var ID 'BCAST_VAL') for this execution of the graph
  MPIX_Graph_set_param(&graph, BCAST_VAL, <neighbor exchange value>);

  // Create deferred operation token for parameterized graph
  MPIX_Graph_def(graph, info, &token);

  //  Create an [inactive] request for the graph token
  MPIX_Execute_init(token, info, MPIX_OFFLOAD_CUDA, &cuda_stream, &req);

  // Release token for parameterized graph
  MPIX_Token_free(&token);

  // Start execution of graph
  MPI_Start(&req);

  <compute current timestep>

  // Conclude graph's operations, if it's not finished yet
  MPI_Wait(&req, &status);
} // timestep loop

// Free graph
MPIX_Graph_free(&graph);
```

Listing 3. – Data-dependent communication operations

This use case shows a fairly straightforward overlap of compute and communication. If the # of neighbor exchanges didn't depend on the the information being broadcast, this could be done with nonblocking operations today. However, the runtime data dependency makes this impossible with today's calls, without waiting for the broadcast to complete before issuing the sendrecv calls.

7.2. Use Case #2: Asynchronously Allocate and Receive Unknown Message

This example code demonstrates receiving an unknown-sized message in the background, with the buffer allocated asynchronously as well.

Although `MPI_Iprobe` and `MPI_Irecv` provide partial functionality needed for this use case, they can't provide the 'future' values needed (for the message size and allocated buffer), the necessary dependencies between operations, or the asynchronous memory allocation. All of these features must be present to allow this sequence of operations to execute independently of the application.

```
MPIX_Graph graph;
MPIX_Token token;
MPI_Request req;
enum {MSG_LEN, MSG_BUF_PTR}; // Graph variable IDs

// Create graph
MPIX_Graph_create(session, info, &graph);

// Create a reference parameter graph variable to alias the graph variable to an application buffer pointer
MPIX_Graph_var_create(&graph, MSG_BUF_PTR, MPIX_GRAPH_VAR_REF_PARAM, MPI_AINT);

// Create a local graph variable to hold the message size
MPIX_Graph_var_create(&graph, MSG_LEN, MPIX_GRAPH_VAR_LOCAL, MPI_COUNT);

// Receive the buffer size into a graph variable
unsigned size_recv_op_id = 0;
MPIX_Recv_def(MSG_LEN, 1, MPI_COUNT, MPI_ANY_SOURCE, MPI_ANY_TAG, comm, MPI_STATUS_IGNORE, &token);
MPIX_Graph_add(&graph, &token, &size_recv_op_id, MPIX_DEP_NONE);

// Allocate memory for the message, using the 'future' value for the message size and returning a future for the buffer pointer
unsigned buf_alloc_op_id = 0;
MPIX_Alloc_mem_def(MSG_LEN, MPI_INFO_NULL, MSG_BUF_PTR, &token);
MPIX_Graph_add(&graph, &token, &buf_alloc_op_id, size_recv_op_id);

// Receive the actual message
unsigned msg_recv_op_id = 0;
MPIX_Recv_def(MSG_BUF_PTR, MSG_LEN, MPI_BYTE, MPI_ANY_SOURCE, MPI_ANY_TAG, comm, MPI_STATUS_IGNORE, &token);
MPIX_Graph_add(&graph, &token, &msg_recv_op_id, buf_alloc_op_id);

// Alias graph parameter for message buffer to application pointer to buffer
void *msg_buf = NULL;
MPIX_Graph_set_param(&graph, MSG_BUF_PTR, &msg_buf);

// Create deferred operation token for parameterized graph
MPIX_Graph_def(graph, info, &token);

// Free graph
MPIX_Graph_free(&graph);
```

```
// Create an [inactive] request for the graph token
MPIX_Execute_init(token, info, MPIX_OFFLOAD_PTHREAD, NULL, &req);

// Release token for parameterized graph
MPIX_Token_free(&token);

// Start execution of graph
MPI_Start(&req);

<compute>

// Conclude graph's operations, if it's not finished yet
MPI_Wait(&req, &status);

// Use the message buffer
<use msg_buf>
```

Listing 4. – Asynchronously allocate space for and receive message of unknown length

7.3. Use Case #3: Prefetch compressed data from file and broadcast to other ranks

This example code demonstrates asynchronously pre-fetching a compressed file on rank 0, decompressing it, and broadcasting the resulting buffer to the other ranks.

This use case demonstrates the power of offloading operations to a "very smart NIC", one which can run both communication and I/O operations asynchronously from the application. The code below offloads file I/O, memory allocation, invoking a user callback, and communication, allowing the application to continue to perform computation in the foreground while all of the data movement and preparation operations for the next timestep occur in the background.

```
MPIX_Graph graph;
MPIX_Token token;
MPI_Request req;
enum {FILE_NAME, BCAST_COMM, BUF_LEN, BUF_PTR, // Graph parameter variable IDs
      FILE_HANDLE, FILE_SIZE, COMP_BUF_PTR};   // Graph variable IDs

// Create graph
MPIX_Graph_create(session, info, &graph);

// Create a reference parameter graph variable to hold the decompressed buffer size and pointer
MPIX_Graph_var_create(&graph, BUF_LEN, MPIX_GRAPH_VAR_REF_PARAM, MPI_COUNT);
MPIX_Graph_var_create(&graph, BUF_PTR, MPIX_GRAPH_VAR_REF_PARAM, MPI_AINT);

// Create a local graph variable to hold the communicator to use for broadcasting data within the graph
MPIX_Graph_var_create(&graph, BCAST_COMM, MPIX_GRAPH_VAR_LOCAL, MPI_LOCAL_TYPE_COMM);

// Rank 0 does the file I/O and bcasts the decompressed data
if (0 == rank) {
  // Create a value parameter graph variable for the filename
  MPIX_Graph_var_create(&graph, FILE_NAME, MPIX_GRAPH_VAR_VAL_PARAM, MPI_LOCAL_TYPE_STRING);

  // Create a local graph variable to hold the file handle
```

```c
MPIX_Graph_var_create(&graph, FILE_HANDLE, MPIX_GRAPH_VAR_LOCAL, MPI_LOCAL_TYPE_FILE);

// Open the file
unsigned file_open_op_id = 0;
MPIX_File_open_def(MPI_COMM_SELF, FILE_NAME, MPI_MODE_RDONLY | MPI_MODE_SEQUENTIAL, MPI_INFO_NULL, FILE_HANDLE, &token);
MPIX_Graph_add(&graph, &token, &file_open_op_id, MPIX_DEP_NONE);

// Create a local graph variable to hold the file's size
MPIX_Graph_var_create(&graph, FILE_SIZE, MPIX_GRAPH_VAR_LOCAL, MPI_OFFSET);

// Get the file's size
unsigned file_size_op_id = 0;
MPIX_File_get_size_def(FILE_HANDLE, FILE_SIZE, &token);
MPIX_Graph_add(&graph, &token, &file_size_op_id, file_open_op_id);

// Create a local graph variable to hold the pointer to the buffer with compressed data
MPIX_Graph_var_create(&graph, COMP_BUF_PTR, MPIX_GRAPH_VAR_LOCAL, MPI_AINT);

// Allocate buffer for compressed data
unsigned alloc_comp_buf_op_id = 0;
MPIX_Alloc_mem_def(FILE_SIZE, MPI_INFO_NULL, COMP_BUF_PTR, &token);
MPIX_Graph_add(&graph, &token, &alloc_comp_buf_op_id, file_size_op_id);

// Read compressed data
unsigned file_read_op_id = 0;
MPIX_File_read_def(FILE_HANDLE, COMP_BUF_PTR, FILE_SIZE, MPI_BYTE, MPI_STATUS_IGNORE, &token);
MPIX_Graph_add(&graph, &token, &file_read_op_id, alloc_comp_buf_op_id);

// Close file
// NOTE:  Concurrent with decompressing data
MPIX_File_close_def(FILE_HANDLE, &token);
MPIX_Graph_add(&graph, &token, NULL, file_read_op_id);

// Decompress data with user callback
// NOTE: Concurrent with closing the file
//
// The decompression function would look something like this:
// void decompress_fn(MPIX_Graph *graph, void *extra_state)
// {
//   MPI_Offset compressed_size = 0;
//   void *compressed_buf = NULL;
//   MPI_Count decompressed_size = 0;
//   void *decompressed_buf = NULL;
//
//   MPIX_Graph_var_get(graph, FILE_SIZE, &compressed_size);
//   MPIX_Graph_var_get(graph, COMP_BUF_PTR, &compressed_buf);
//   <decompression routine>(/*IN:*/compressed_size, /*IN:*/compressed_buf, /*OUT:*/&decompressed_size, /*OUT:*/&decompressed_buf);
//   MPIX_Graph_var_set(graph, BUF_LEN, &decompressed_size);
//   MPIX_Graph_var_set(graph, BUF_PTR, &decompressed_buf);
// }
unsigned decomp_op_id = 0;
MPIX_Op_def(&decompress_fn, NULL, &token);
MPIX_Graph_add(&graph, &token, &decomp_op_id, file_read_op_id);
```

```c

  // Bcast the decompressed buffer size, from rank 0
  unsigned size_bcast_op_id = 0;
  MPIX_Bcast_def(BUF_LEN, 1, MPI_COUNT, 0, BCAST_COMM, &token);
  MPIX_Graph_add(&graph, &token, &size_bcast_op_id, decomp_op_id);

  // Bcast the decompressed buffer, from rank 0
  MPIX_Bcast_def(BUF_PTR, BUF_LEN, MPI_BYTE, 0, BCAST_COMM, &token);
  MPIX_Graph_add(&graph, &token, NULL, size_bcast_op_id);
} // end (0 == rank)
else { // (0 != rank)
  // Receive the decompressed buffer size bcast from rank 0
  unsigned size_bcast_op_id = 0;
  MPIX_Bcast_def(BUF_LEN, 1, MPI_COUNT, 0, BCAST_COMM, &token);
  MPIX_Graph_add(&graph, &token, &size_bcast_op_id, MPIX_DEP_NONE);

  // Allocate buffer for decompressed data
  unsigned alloc_decomp_buf_op_id = 0;
  MPIX_Alloc_mem_def(BUF_LEN, MPI_INFO_NULL, BUF_PTR, &token);
  MPIX_Graph_add(&graph, &token, &alloc_decomp_buf_op_id, size_bcast_op_id);

  // Receive the decompressed buffer bcast from rank 0
  MPIX_Bcast_def(BUF_PTR, BUF_LEN, MPI_BYTE, 0, BCAST_COMM, &token);
  MPIX_Graph_add(&graph, &token, NULL, alloc_decomp_buf_op_id);
} // end (0 != rank)

// Set communicator for bcast operations in graph
MPI_Comm graph_comm = MPI_COMM_WORLD;
MPIX_Graph_var_set(&graph, BCAST_COMM, &graph_comm);

//  Prefetch next timestep's data from disk while current timestep is computing
while(<more timesteps>) {
  // Rank 0 does the file I/O and bcasts the decompressed data
  if (0 == rank) {
    const char *filename = <filename for timestep 'n'>;
    MPIX_Graph_set_param(&graph, FILE_NAME, &filename);
  }

  // Alias graph parameters for decompressed data buffer info to application variables
  void *buf_ptr = NULL;
  MPI_Count buf_len = 0;
  MPIX_Graph_set_param(&graph, BUF_PTR, &buf_ptr);
  MPIX_Graph_set_param(&graph, BUF_LEN, &buf_len);

  // Create deferred operation token for parameterized graph
  MPIX_Graph_def(graph, info, &token);

  // Create an [inactive] request for the graph token
  MPIX_Execute_init(token, info, MPIX_OFFLOAD_NITRO, NULL, &req);

  // Release token for parameterized graph
  MPIX_Token_free(&token);

  // Start execution of graph
  MPI_Start(&req);

```

```
  <compute>

  // Conclude graph's operations, if it's not finished yet
  MPI_Wait(&req, &status);

  <swap data buffers for next timestep>
}

// Free graph
MPIX_Graph_free(&graph);
```

Listing 5. – Asynchronously prefetch a compressed file on rank 0 and broadcast the decompressed buffer to other ranks

7.4. Use Case #4: Offload Collective I/O For Writing Checkpoint File

This example code demonstrates asynchronously writing a checkpoint, using collective I/O.

This use case again shows fully overlapping I/O with compute. In particular, this example includes concurrently executing collective I/O operations, and using the “join” operator to create a single dependency for the file close operation to depend on.

```
MPIX_Graph graph;
MPIX_Token token;
MPI_Request req;
enum {FILE_NAME, FILE_COMM,  // Graph parameter variable IDs for file
      BUF1_OFFSET, BUF1_PTR, BUF1_COUNT, BUF1_TYPE,  // Graph parameter
       ↪  variable IDs for buffer #1
      BUF2_OFFSET, BUF2_PTR, BUF2_COUNT, BUF2_TYPE,  // Graph parameter
       ↪  variable IDs for buffer #2
      BUF3_OFFSET, BUF3_PTR, BUF3_COUNT, BUF3_TYPE,  // Graph parameter
       ↪  variable IDs for buffer #3
      FILE_HANDLE};          // Graph local variable IDs for file

// Create graph
MPIX_Graph_create(session, info, &graph);

// Create a value parameter graph variable for the filename and file's
 ↪  communicator
MPIX_Graph_var_create(&graph, FILE_NAME, MPIX_GRAPH_VAR_VAL_PARAM,
 ↪  MPI_LOCAL_TYPE_STRING);
MPIX_Graph_var_create(&graph, FILE_COMM, MPIX_GRAPH_VAR_VAL_PARAM,
 ↪  MPI_LOCAL_TYPE_COMM);

// Create a value parameter graph variables for the three buffers to write
MPIX_Graph_var_create(&graph, BUF1_OFFSET, MPIX_GRAPH_VAR_VAL_PARAM,
 ↪  MPI_OFFSET);
MPIX_Graph_var_create(&graph, BUF1_PTR, MPIX_GRAPH_VAR_VAL_PARAM, MPI_AINT);
MPIX_Graph_var_create(&graph, BUF1_COUNT, MPIX_GRAPH_VAR_VAL_PARAM,
 ↪  MPI_COUNT);
MPIX_Graph_var_create(&graph, BUF1_TYPE, MPIX_GRAPH_VAR_VAL_PARAM,
 ↪  MPI_LOCAL_TYPE_DATATYPE);

MPIX_Graph_var_create(&graph, BUF2_OFFSET, MPIX_GRAPH_VAR_VAL_PARAM,
 ↪  MPI_OFFSET);
MPIX_Graph_var_create(&graph, BUF2_PTR, MPIX_GRAPH_VAR_VAL_PARAM, MPI_AINT);
MPIX_Graph_var_create(&graph, BUF2_COUNT, MPIX_GRAPH_VAR_VAL_PARAM,
 ↪  MPI_COUNT);
```

```
MPIX_Graph_var_create(&graph, BUF2_TYPE, MPIX_GRAPH_VAR_VAL_PARAM, MPI_LOCAL_TYPE_DATATYPE);

MPIX_Graph_var_create(&graph, BUF3_OFFSET, MPIX_GRAPH_VAR_VAL_PARAM, MPI_OFFSET);
MPIX_Graph_var_create(&graph, BUF3_PTR, MPIX_GRAPH_VAR_VAL_PARAM, MPI_AINT);
MPIX_Graph_var_create(&graph, BUF3_COUNT, MPIX_GRAPH_VAR_VAL_PARAM, MPI_COUNT);
MPIX_Graph_var_create(&graph, BUF3_TYPE, MPIX_GRAPH_VAR_VAL_PARAM, MPI_LOCAL_TYPE_DATATYPE);

// Create a local graph variable to hold the file handle
MPIX_Graph_var_create(&graph, FILE_HANDLE, MPIX_GRAPH_VAR_LOCAL, MPI_LOCAL_TYPE_FILE);

// Create the file
unsigned file_create_op_id = 0;
MPIX_File_open_def(FILE_COMM, FILE_NAME, MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, FILE_HANDLE, &token);
MPIX_Graph_add(&graph, &token, &file_create_op_id, MPIX_DEP_NONE);

// Collectively write all three buffers, concurrently
unsigned file_write_buf_op_ids[3] = {0, 0, 0};
MPIX_File_write_at_all_c_def(FILE_HANDLE, BUF1_OFFSET, BUF1_PTR, BUF1_COUNT, BUF1_TYPE, MPI_STATUS_IGNORE, &token);
MPIX_Graph_add(&graph, &token, &file_write_buf_op_ids[0], file_create_op_id);

MPIX_File_write_at_all_c_def(FILE_HANDLE, BUF2_OFFSET, BUF2_PTR, BUF2_COUNT, BUF2_TYPE, MPI_STATUS_IGNORE, &token);
MPIX_Graph_add(&graph, &token, &file_write_buf_op_ids[1], file_create_op_id);

MPIX_File_write_at_all_c_def(FILE_HANDLE, BUF3_OFFSET, BUF3_PTR, BUF3_COUNT, BUF3_TYPE, MPI_STATUS_IGNORE, &token);
MPIX_Graph_add(&graph, &token, &file_write_buf_op_ids[2], file_create_op_id);

// Create a 'join' operation ID, for all writes
unsigned join_writes_op_id = 0;
MPIX_Graph_join(&graph, 3, file_write_buf_op_ids, &join_writes_op_id);

// Close file
// NOTE:  Depends on join of all concurrent collective writes
MPIX_File_close_def(FILE_HANDLE, &token);
MPIX_Graph_add(&graph, &token, NULL, join_writes_op_id);

//  Compute, then write checkpoint, overlapped with next compute
boolean checkpoint_io_started = FALSE;
while(<more timesteps>) {
  <compute>

  // Make certain that previous checkpoint has completed
  if (checkpoint_io_started) {
    MPI_Wait(&req, &status);
    checkpoint_io_started = FALSE;
  }

  // Write checkpoint
  if (<time to make a checkpoint>) {
    const char *filename = <checkpoint filename>;
```

```
    MPI_Comm file_comm = <communicator for file I/O>;

    // Set filename & communicator
    MPIX_Graph_set_param(&graph, FILE_NAME, &filename);
    MPIX_Graph_set_param(&graph, FILE_COMM, &file_comm);

    // Set buffer info
    MPIX_Graph_set_param(&graph, BUF1_OFFSET, &buf1_off[rank]);
    MPIX_Graph_set_param(&graph, BUF1_PTR,    &buf1_ptr[rank]);
    MPIX_Graph_set_param(&graph, BUF1_COUNT,  &buf1_count[rank]);
    MPIX_Graph_set_param(&graph, BUF1_TYPE,   &buf1_type[rank]);

    MPIX_Graph_set_param(&graph, BUF2_OFFSET, &buf2_off[rank]);
    MPIX_Graph_set_param(&graph, BUF2_PTR,    &buf2_ptr[rank]);
    MPIX_Graph_set_param(&graph, BUF2_COUNT,  &buf2_count[rank]);
    MPIX_Graph_set_param(&graph, BUF2_TYPE,   &buf2_type[rank]);

    MPIX_Graph_set_param(&graph, BUF3_OFFSET, &buf3_off[rank]);
    MPIX_Graph_set_param(&graph, BUF3_PTR,    &buf3_ptr[rank]);
    MPIX_Graph_set_param(&graph, BUF3_COUNT,  &buf3_count[rank]);
    MPIX_Graph_set_param(&graph, BUF3_TYPE,   &buf3_type[rank]);

    // Create deferred operation token for parameterized graph
    MPIX_Graph_def(graph, info, &token);

    // Create an [inactive] request for the graph token
    MPIX_Execute_init(token, info, MPIX_OFFLOAD_NITRO, NULL, &req);

    // Release token for parameterized graph
    MPIX_Token_free(&token);

    // Start execution of graph
    MPI_Start(&req);

    // Indicate that the I/O has started
    checkpoint_io_started = TRUE;

    <swap data buffers for next timestep>
  }
}

// Free graph
MPIX_Graph_free(&graph);
```

Listing 6. – Asynchronously write checkpoint file with collective I/O

7.5. Use Case #5: Data Dependent Asynchronous Communication

This example code demonstrates conditionally executing communication operations, depending on data values across the application.

This use case creates a reusable parameterized graph that is invoked to overlap with computation. It uses an asynchronous data dependency to control execution of a sub-graph of asynchronous communication, all of which can be offloaded or executed in the background.

Note that executing this graph is not the same as passing parameters to a subroutine in the application that invokes asynchronous operations. Such a subroutine would need to block on the allreduce operations in order to compare the results. However, the 'IfElse' operation used in the graph is asynchronously executed in the same manner as the asynchronous communication operations and is managed by MPI, not the application. A similar result to the graph execution could be created by running such a subroutine in an application thread, but that would prevent the MPI implementation from seeing the 'big picture' of all the planned operations and reduce opportunities for optimizing the operations in the graph.

```
MPIX_Graph graph, subgraph;
MPIX_Token token;
MPI_Request req;
enum {GRAPH_COMM, VAR_A, VAR_B, SEND_BUF, RECV_BUF,  // Graph parameter
 ↪  variable IDs for conditional communication
      COMP_VAL};    // Graph parameter variable IDs for comparison

// Create main graph
MPIX_Graph_create(session, info, &graph);

// Create value parameter graph variables for graph communications
MPIX_Graph_var_create(&graph, GRAPH_COMM, MPIX_GRAPH_VAR_VAL_PARAM,
 ↪  MPI_LOCAL_TYPE_COMM);

// Create value parameter graph variables for controlling data exchange
MPIX_Graph_var_create(&graph, VAR_A, MPIX_GRAPH_VAR_VAL_PARAM, MPI_INT);
MPIX_Graph_var_create(&graph, VAR_B, MPIX_GRAPH_VAR_VAL_PARAM, MPI_INT);

// Create a value parameter graph variables for the send & recv buffers
MPIX_Graph_var_create(&graph, SEND_BUF, MPIX_GRAPH_VAR_VAL_PARAM, MPI_AINT);
MPIX_Graph_var_create(&graph, RECV_BUF, MPIX_GRAPH_VAR_VAL_PARAM, MPI_AINT);

// Concurrently perform all reduce operations on 'A' and 'B'
unsigned allreduce_op_ids[2];
MPIX_Allreduce_def(MPI_IN_PLACE, VAR_A, 1, MPI_INT, MPI_MAX, GRAPH_COMM,
 ↪  &token);
MPIX_Graph_add(&graph, &token, NULL, &allreduce_op_ids[0]);

MPIX_Allreduce_def(MPI_IN_PLACE, VAR_B, 1, MPI_INT, MPI_MAX, GRAPH_COMM,
 ↪  &token);
MPIX_Graph_add(&graph, &token, NULL, &allreduce_op_ids[1]);

// Create a 'join' operation ID, for all writes
unsigned join_reduces_op_id = 0;
MPIX_Graph_join(&graph, 2, allreduce_op_ids, &join_reduces_op_id);

// Create a reference parameter graph variable for comparison result
MPIX_Graph_var_create(&graph, COMP_VAL, MPIX_GRAPH_VAR_REF_PARAM, MPI_INT);

// Compare 'A' and 'B' values from allreduce operations
unsigned compare_op_id = 0;
MPIX_Graph_var_reduce(&graph, VAR_A, VAR_B, MPIX_LESS_THAN_OR_EQUAL, COMP_VAL,
 ↪  &compare_op_id, join_reduces_op_id);

// Create sub-graph, for if/else
MPIX_Graph_create(session, info, &subgraph);

// Sendrecv operation for if/else
MPIX_Sendrecv_def(SEND_BUF, 2, MPI_FLOAT, (rank + 1) % num_ranks, 0, RECV_BUF,
 ↪  2, MPI_FLOAT, (num_ranks + (rank - 1)) % num_ranks, 0, GRAPH_COMM,
 ↪  MPI_STATUS_IGNORE);
MPIX_Graph_add(&subgraph, &token, NULL, MPIX_DEP_NONE);
```

```
// Execute subgraph if 'A' <= 'B'
MPIX_Graph_IfElse(&graph, COMP_VAL, &subgraph, MPIX_GRAPH_NULL, NULL,
 ↪  compare_op_id);

// Free subgraph
MPIX_Graph_free(&subgraph);

// Set communicator for all graph executions
MPI_Comm graph_comm = <communicator>;
MPIX_Graph_set_param(&graph, GRAPH_COMM, &graph_comm);

int comp_val = 0;
while(<more timesteps>) {
  // Check results of previous timestep's graph execution
  if (<timestep > 0>) {
    // Make certain that graph has completed
    MPI_Wait(&req, &status);

    // Check if graph's condition was met
    if(comp_val) {
      <use value in recv_buf>
      comp_val = 0;
    }
  }

  // Application variables on each rank, for graph
  int a = compute_a(timestep);
  int b = compute_b(timestep);
  float sendbuf[2], recvbuf[2];

  <init sendbuf & recvbuf>

  // Set parameters for graph on each rank and execute it
  MPIX_Graph_set_param(&graph, VAR_A, &a);
  MPIX_Graph_set_param(&graph, VAR_B, &b);
  MPIX_Graph_set_param(&graph, SEND_BUF, &sendbuf);
  MPIX_Graph_set_param(&graph, RECV_BUF, &recvbuf);
  MPIX_Graph_set_param(&graph, COMP_VAL, &comp_val);

  // Create deferred operation token for parameterized graph
  MPIX_Graph_def(graph, info, &token);

  // Create an [inactive] request for the graph token
  MPIX_Execute_init(token, info, MPIX_OFFLOAD_PTHREAD, NULL, &req);

  // Release token for parameterized graph
  MPIX_Token_free(&token);

  // Start execution of graph
  MPI_Start(&req);

  <compute>

}

// Check results of last timestep's graph execution
if (<timestep > 0 >) {
```

```
  // Make certain that graph has completed
  MPI_Wait(&req, &status);

  // Check if graph's condition was met
  if(comp_val)
    <use value in recv_buf>
}

// Free graph
MPIX_Graph_free(&graph);
```

Listing 7. – Data Dependent Asynchronous Operations

A. Appendix: An Abstract Data Movement Machine

If there was a chip that provided "assembly" instructions that could be used to implement the MPI operations described above, what would those instructions be?

A.1. Data Containers

A "container" is an abstraction around a particular sequence of bytes and provides the base object for data operations.

A container is one of four things:

- *Local* memory – memory at this MPI process, which could be attached to a CPU, GPU, FPGA, etc.
- *Remote* memory – memory at another MPI process, which could (also) be attached to a CPU, GPU, FPG, etc.
- *Abstract machine* memory – memory in one of the abstract 'data movement machines' being described here. (More details on this type of memory are outlined below, and are called 'variables' there)
- *Storage* memory – a file on an SSD, HDD, tape, a cloud object, an RMA window, etc. This form of memory is *passive* – it doesn't have an MPI process managing it.

Comment [QK4]: Elaborate on Ryan and my discussion today about these types of memory / containers being associated with a process space, except for storage, etc.

Need a better word than 'abstract machine'. Maybe "virtual", "transient", "temporary", or "intragraph" instead?

The virtual machine must provide mechanisms to: create new, access existing, release access to, and delete these types of containers. (i.e. 'create', 'open', 'close', and 'delete' container operations)

A.2. Container Operations

The fundamental operations for containers in the virtual machine are abstractions for distributed "copy data", "reduce", "compare", and "synchronize". These operations provide the core capabilities necessary to implement MPI operations, as well as other similar data-oriented frameworks, and are described below.

A.2.1. Abstract Copy Operations

The "abstract copy" ("ACOPY") operation copies bytes from a list of 1+ source locations to a list of 1+ destination locations. A location is defined as a tuple of {container, offset, length}. The offset and length for a location tuple describes a byte range in a container.

The ACOPY operation copies bytes from source containers to destination containers, in the order given in each location list. This copy operation does not require that the lengths of each byte range be equal in the lists, only that the total number of bytes described by the source location list is equal to the total number of bytes described by the destination location list. The ACOPY operation also does not require that byte ranges are non-overlapping, non-repeating, or sorted in any particular order, although that might be imposed by a higher-level abstraction layer (like MPI!).

This ACOPY operation allows for arbitrary scatter-gather operations between any type of data containers. Again, higher-level layers (like MPI) may impose various constraints on the location lists, in order to improve performance or other aspects of execution.

A.2.2. Abstract Reduce Operations

Distributed arithmetic operations on data container locations are analogous to the existing MPI 'reduce' operation [9] . As such, any type of container (local, remote, abstract machine, or storage) may be used as inputs to or the output from an abstract reduce ("AREDUCE") operation:

$output = reduce(op, input_0, input_1, ...)$

Valid mathematical operations for the *op* in an AREDUCE operation must be both commutative and associative. Examples include addition, multiplication, logical AND and OR, binary AND and OR, union and intersection on sets, greatest common divisor, least common multiple, minimum, and maximum.

The AREDUCE operation allows for arbitrary reduce operations between locations in any type of data container. Again, higher-level layers (like MPI) may impose constraints on the location of inputs, in order to improve performance or other aspects of execution.

A.2.3. Abstract Comparison Operations

Comparison operations on container locations don't have a direct analog to an existing MPI operation, but are closest to a distributed consensus operation[10]. The abstract comparison ("ACOMPARE") operation takes a single primary operand ("*primary*"), a comparison operation ("*op*"), a set of n secondary operands ("$secondary_0..secondary_{n-1}$"), and a threshold value ("*threshold*") as inputs and produces a boolean output value that is true if more than $threshold$ of the $(primary, op, secondary)$ comparisons are true and false if the $threshold$ value is not reached.

$output = compare(threshold, primary, op, secondary_0, secondary_1, ...)$

Valid comparison operations for the *op* in an ACOMPARE operation must produce a binary true or false value for comparing the $primary$ operand to each of the $secondary$ operands. Examples include $==, \neq, <, \leq, >, \geq$, and possibly others.

This ACOMPARE operation allows for arbitrary comparison operations between locations in any type of memory container. Again, higher-level layers (like MPI) may impose constraints on the location of inputs, in order to improve performance or other aspects of execution.

A.2.4. Abstract Synchronization Operations

Synchronization operations on container locations are analogous to the existing MPI 'barrier' operation or a synchronous send+receive pair. Abstract synchronization ("ASYNCHRONIZE") operations take two forms:

- *Data* synchronization – Synchronize 2+ processes to be certain that some/all of the processes have the same data in a container location. [11]
- *Execution* synchronization – Synchronize 2+ processes to be certain that some/all of the processes have/haven't reached the same point in executing an application. [12]

Need better wording here!

A.2.5. Constants

Constant ('literal') values, such as '3' or '2.1', can be used in the following operations:

Comment [QK5]: Interesting variations on barrier could include: a "threshold barrier" operations ("at least 6 of ¡n¿ processes must have reached this barrier

- *ACOPY* – a constant may be used as a source location.

[9]Note that some existing MPI operations or reduce operators must be built up from multiple ACOPY or AREDUCE operations. MPI all-reduce and scan operations and the MPI_MINLOC and MPI_MAXLOC reduce operations are examples of these.

[10]https://en.wikipedia.org/wiki/Consensus_(computer_science)

[11]A pair of synchronous send+receive MPI operations is an example of data synchronization.

[12]The MPI 'barrier' operation is an example of execution synchronization.

- *AREDUCE* – a constant may be used as an input.
- *ACOMPARE* – a constant may be used as either a primary or secondary value.

Values for constants used in graph operations are captured during graph definition.

A.3. Graphs

A "virtual task graph" with an execution unit that understands dependencies between 0+ "parent" tasks and 0+ "child" tasks, along with "control flow" operations, and "virtual" memory/registers for the graph execution unit best describes the next components of the "abstract data movement machine".

A.3.1. Graph Task Execution Unit

The "graph task execution unit" understands the following types of "task operations" as nodes in the graph:

- *Container* operations – ACOPY, AREDUCE, ACOMPARE and ASYNCHRONIZE operations on container locations, as well as container create/access/release/delete operations.
- *User* operation – an operation that allows an application callback as an operation in the graph.
- *Subgraph* operation – an entire graph as an "operation" in the current graph.

Disallow 'self' as valid "subgraph op" for now, but consider a mechanism for recursion in the future.

A.3.2. Graph Dependency Unit

The "graph dependency unit" manages dependencies between graph tasks, i.e. the edges in the graph. There are several types of dependencies, but they fundamentally are a mechanism for ordering execution of graph operations.

Task and Data Dependencies There are two types of dependencies for graph operations: task and data. Tasks dependencies are explicitly created by an application, by specifying parent operations as 'input' dependencies for an operation, all of which must complete before the operation can be executed. Data dependencies are indirectly created by an application, by using data or values produced by an earlier graph operation as an input value to another operation.

Task operations (A.3.1) each have a single 'input' task dependency (i.e. one parent) and a single 'output' task depedency (i.e. one child), as shown in Fig. 1. When their input dependency has been fulfilled (i.e. their parent task has completed), the task may be scheduled for execution. Correspondingly, when the task completes, its output dependency is fulfilled and its child task may be scheduled for execution. More complicated dependencies between graph tasks can be constructed with "organizational" dependencies, described below.

Each task operation may also have 0+ input data dependencies and 0+ output data dependencies. Data dependencies are implicitly created between operations when data or values produced by one operation (i.e. a return value or an "out" parameter) is consumed by another operation (i.e. used as an "in" or "in/out" parameter). There are two kinds of data dependencies:

- *Strong* data dependencies – provide or create access to a container (i.e. a file handle, memory buffer, etc) and can be represented as a "future" value.
- *Weak* data dependencies – access or modification of the contents of container (i.e. ACOPY, AREDUCE, ACOMPARE or ASYNCHRONIZE operations on locations within a container).

Strong data dependencies can be reliably tracked between task operations, therefore a dependency between the operation that generates a future value and any operations that use that value doesn't need to be explicitly defined by an application. However, weak dependencies can have ambiguous ordering that only an application

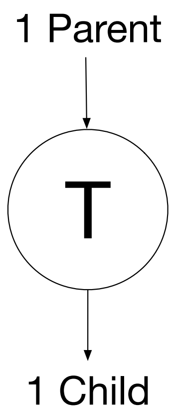

Figure 1 – Task dependencies

programmer understands. Operations that rely on ordered access to or modification of the contents of a container must therefore explicitly define that ordering with task or organizational dependencies between the operations.

Organizational Dependencies “Organizational” dependencies don’t actually perform any work, they just provide “connections” in the graph to indicate and manage ordering of task operations. The graph dependency unit understands the following types of organizational dependencies:

- *Unconditional* dependencies – Unconditional dependencies connect 0+ input dependencies (“parents”) to 0+ output dependencies (“children”) without regard to any other state for the graph. All of the input dependencies for an unconditional organizational operation (if any are defined) must complete before any of the output dependencies may start execution. The general form of an unconditional dependency is represented in Fig. 2.

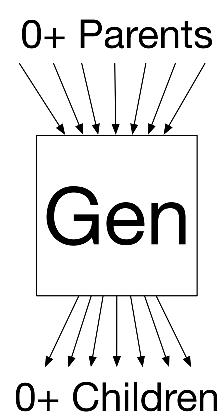

Figure 2 – General form of unconditional organizational graph dependencies

  Common specialized forms of unconditional dependencies are root, fan-in, fan-out, and leaf, as shown in Fig. 3.

- *Conditional* dependencies – Conditional dependencies connect 0+ input dependencies (“parents”) to 0+ output dependencies (“children”) and have a condition that must be met to fulfill the output dependency. Two forms of conditional organizational dependencies are:

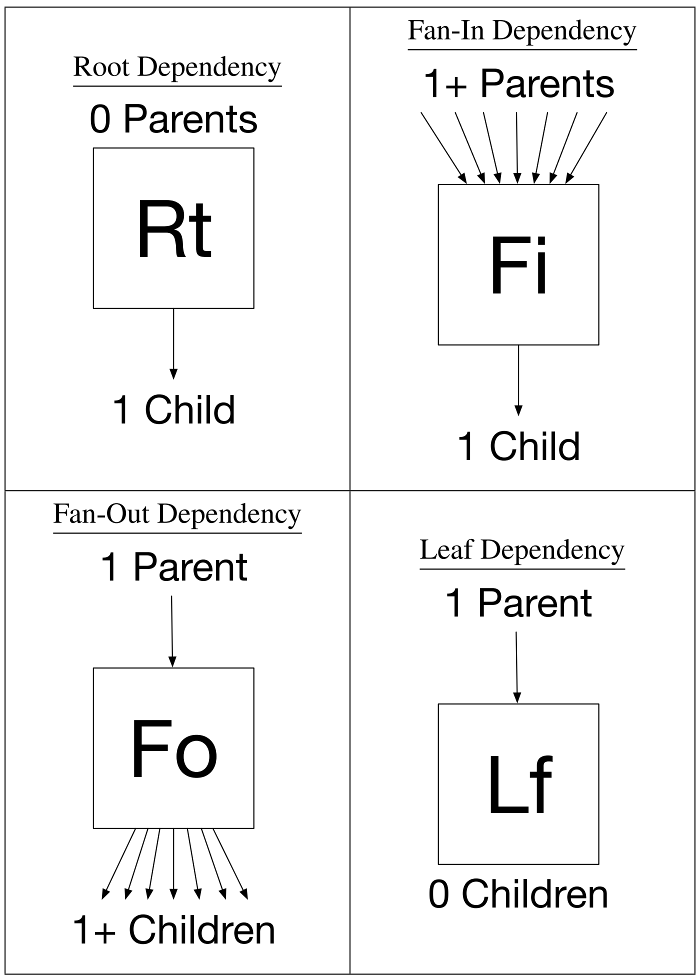

Figure 3 – Common forms of unconditional organizational graph dependencies

– *Consensus* dependencies – Consensus dependencies are fulfilled when a (user-defined) threshold of their input dependencies are fulfilled, providing a "some but not all" or "enough" form of dependency, represented in Fig. 4.

Comment [QK6]: Should the threshold be constant or allowed to depend on a graph variable?

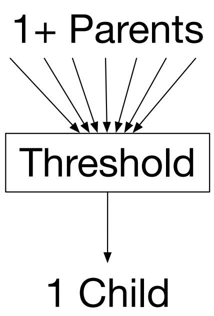

Figure 4 – Threshold organizational graph dependency

– *Branch* dependencies – Branch dependencies connect a different output dependency to an input dependency, depending on the state of a graph variable, as shown in Fig. 5.

Note that a graph variable is examined within the branch dependency, which could have weak data dependencies on other concurrently executing operations in the graph that create race conditions when the graph is executed. Application developers can use other organizational dependencies as preludes to the branch dependency to eliminate such race conditions.

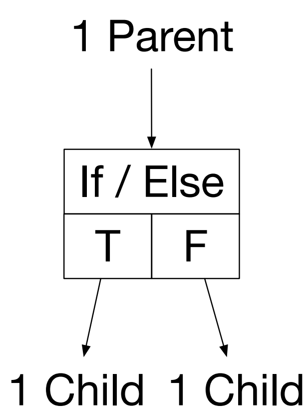

Figure 5 – Branch organizational graph dependency

Graph Control Flow "Control flow" operations also don't perform any work, they just indicate the next task to execute, as if the destination of the control flow operation was a child dependency of another operation. However, control flow operations don't create "full" dependencies in the graph – they are child dependencies of their parent task, but not parent dependencies of their destination task. Control flow operations act similar to a 'goto' operation in a traditional programming language and are shortened to the 'AGOTO' mnemonic in this document and drawn in diagrams as shown in Fig. 6.

The destination of a control flow operation must be a task where all of the destination task's descendants (children, grandchildren, etc) **only** have parent dependencies that can be traced to the destination task. For example, in Fig. 7, tasks 'A', 'B', and 'C' in the top row are not valid destination tasks, but task 'D' in the middle row is, as are tasks 'E', 'F', and 'G' in the bottom two rows.

Infinite loops are possible with AGOTO operations, but can be interrupted by a 'cancel' operation on the graph when it is executing.

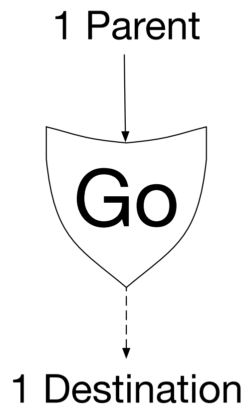

Figure 6 – Control flow operation

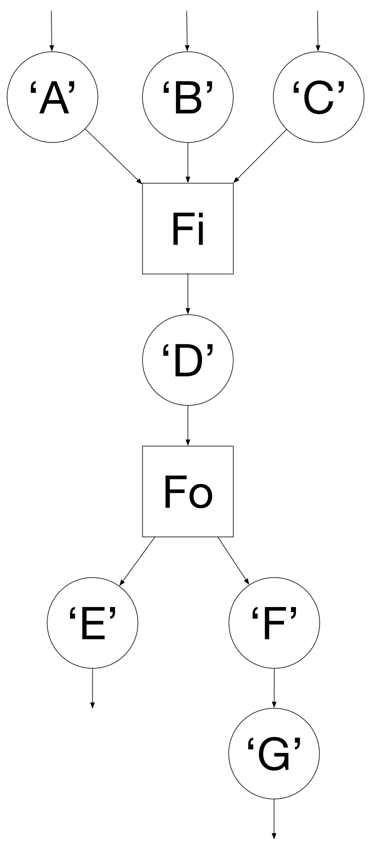

Figure 7 – Example graph for control flow destinations

A.3.3. Graph Construction Rules

Any combination of tasks and organizational dependencies is allowed. Tasks can be connected directly together, as shown in Fig. 8, organizational dependencies can be connected directly together, as shown in Fig. 9, and they can be combined, as shown in Fig. 10.

Tasks, organizational dependencies, and control flow operations can be combined to make data-dependent loop structures in a graph, as shown in Fig. 11.

Tasks, organizational dependencies, subgraphs, and control flow operations can be combined to make complex nested graphs, as shown in Fig. 12.

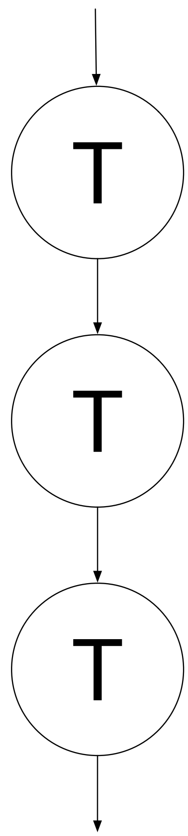

Figure 8 – Connecting multiple task dependencies

A.3.4. Graph Variables

"Graph variables" fill the role of registers and memory for the graph execution virtual machine.

There are three types of graph variables:

- *Value parameter* variable – a variable that is initialized with a value when the graph starts execution.
- *Reference parameter* variable – a variable that is an alias for another container location, set when the graph starts execution.
- *Local* variable – a variable that is initialized to "all zeroes" when the graph starts execution, and is not aliased to a container location.

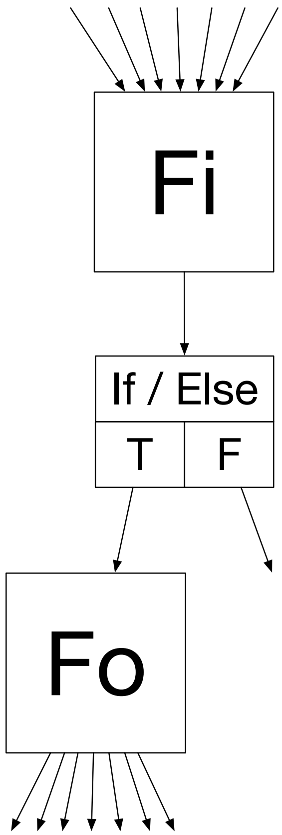

Figure 9 – Connecting multiple organizational dependencies

Variables may be declared at any point during a graph's definition, but assignments for all value or reference parameter variables defined anywhere in the graph must be provided at the time when the graph starts execution.

> The MPI API will need to provide a way to provide values/assign references to parameters when a graph is executed.

> See if it's possible to support "string" types.

The following operations are defined for graph variables:

- *Definition* – declaring a graph variable. Variables are not explicitly deleted, but any resources associated with them are released when a graph terminates.
- *ACOPY* – a graph variable is a valid container location as the source or destination for ACOPY operations.
- *AREDUCE* – a graph variable is valid for both input and output locations for AREDUCE operations.
- *ACOMPARE* – a graph variable is valid for both primary and secondary locations for ACOMPARE operations.
- *ASYNCHRONIZE* – a graph variable is valid for data synchronization ASYNCHRONIZE operations.

'Set' or 'get' operations on graph variables are ACOPY operations between a graph virtual machine container and another location. If the value of a value parameter or local variable is to be retained after its graph terminates, the value must be ACOPY'd from the graph variable before graph termination.

A.4. Streams

A "virtual task stream" with an execution unit that understands asynchronous execution of an ordered sequence of operations, and "virtual" memory/registers for the stream execution unit best describes the next component of the "abstract data movement machine".

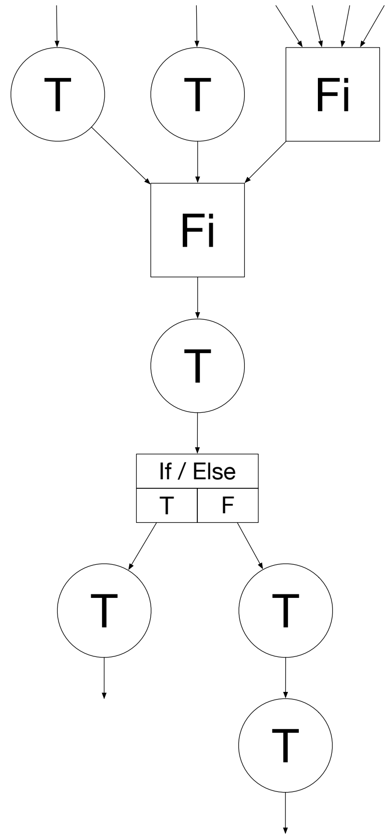

Figure 10 – Combining task and organizational dependencies

A.4.1. Stream Task Execution Unit

Streams operate as FIFO queues to execute operations, similar to a conveyor belt: the first operation in a stream executes to completion and is removed from the stream, then the following operation can be scheduled for execution. The "stream task execution unit" understands the following types of "task operations" as 'nodes' in an ordered sequence:

- *Container* operations – ACOPY, AREDUCE, ACOMPARE and ASYNCHRONIZE operations on container locations, as well as container create/access/release/delete operations.
- *User* operation – an operation that allows an application callback as an operation in the stream.
- *Subgraph* operation – an entire graph as an "operation" in the current stream.

Disallow nesting 'sub-streams' for now, but consider in the future.

A.4.2. Stream Construction Rules

Any combination of tasks and sub-graphs is allowed. Tasks can be inserted as operations in the stream, as shown in Fig. 13 and sub-graphs can be inserted as stream operations as well, as shown in Fig. 14.

A.4.3. Stream Variables

"Stream variables" fill the role of registers and memory for the stream execution virtual machine and are identical to the graph variables defined in section A.3.4 except that they are defined for a stream, not a graph.

Comment [QK7]: Not certain it makes sense to have variables for streams.

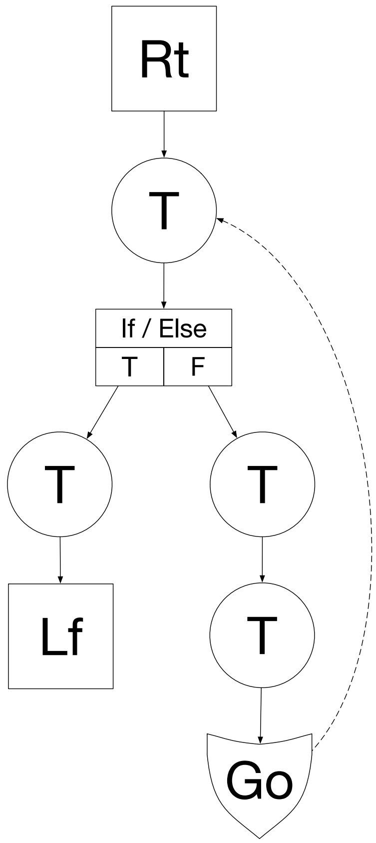

Figure 11 – Combining tasks, organizational dependencies, and control flow operations to create loop in graph

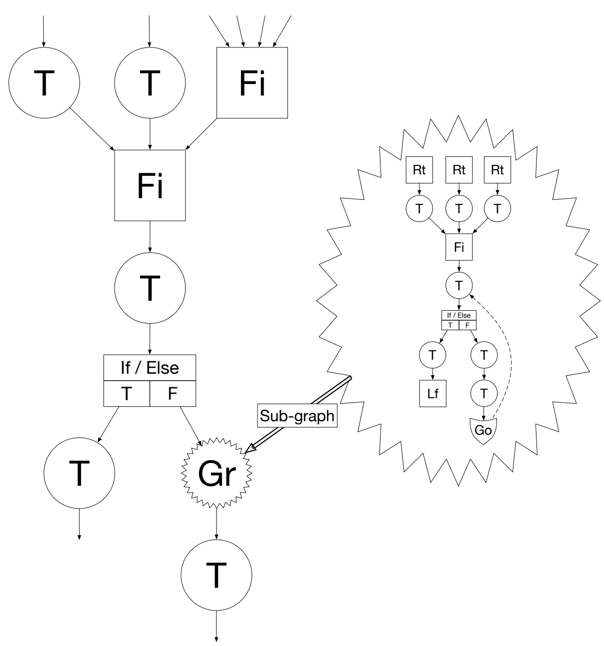

Figure 12 – Combining tasks, organizational dependencies, and sub-graphs

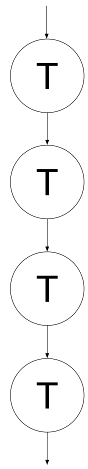

Figure 13 – Tasks in a stream

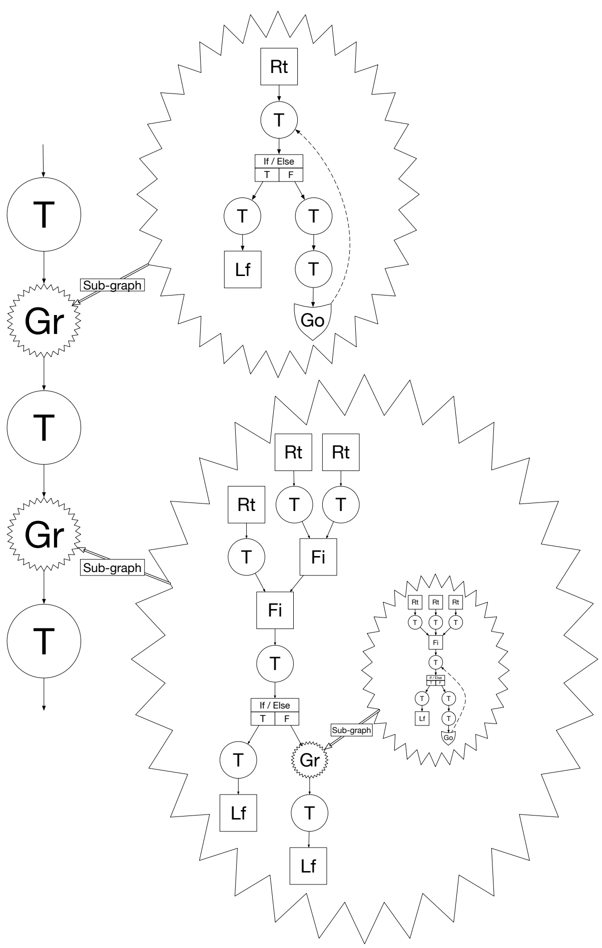

Figure 14 – Tasks and sub-graphs in a stream

A.5. Scopes

B. Appendix: Comparison to proposed Continuations and MPIX_Stream extensions

At first glance, the capabilities described in this document appear similar to the proposed "Continuations" and "MPIX_Stream" extensions..

Add bibrefs for these

Continuations allow a user callback to be attached to an nonblocking MPI operation. Then the continuation callback is invoked when the nonblocking operation completes. Although the capabilities described in this document do include an asynchronous user callback, they are not directly comparable with the user callbacks described in this document. A continuation callback could be executed as part of the progress engine, unless they are explicitly excluded and require that the MPI implementation invoke it when the test or wait operation is performed on a request. However, the user callbacks described in this document are standalone asynchronous operations, fully integrated into the flow of other asynchronous communication and I/O operations.

The MPIX_Stream extensions overlaps some of the ideas presented here, in the area of scheduling operations. However, they are mainly focused on enabling communication with compute endpoints on GPUs and similar hardware components. The asynchronous operations described in this document are essentially orthogonal to them, and both extensions can be integrated into the MPI standard without conflict.

C. Appendix: What this proposal is *not*

Although the capabilities described here may operate best with an MPI implementation that executes them with a background thread or on dedicated hardware, that is not a requirement. The capabilities described here will operate correctly with the typical 'weak' progress model for MPI, just not as efficiently as if they were executing with 'strong' progress. If threads or hardware offload are not used, the 'asynchronous" operations described here may be implemented in the same way as the "nonblocking" operations are currently.

The operations described here are also not designed to provide the capabilities of a computationally-focused task execution framework such as CUDA, etc. Although a call to asynchronously execute an application-provided callback function is provided, it is intended as an "escape hatch" mechanism for short-running operations.

Likewise, the asynchronous operations described here are primarily data movement operations that have minimal performance impact on an application's execution time. A high-quality implementation will schedule asynchronous data movement operations quickly and give up the CPU as soon as possible.

D. Appendix: Details for Asynchronous Operations Option #2

Add details from lab notebook here.