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 may be entered for nonblocking operations to make 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.

Contents

List of Figures

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.

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 application, 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_Irecv(..., &req[0])`
`MPI_Isend(..., &req[1])`	`MPI_Irecv(..., &req[1])`
`...`	`...`
`MPI_Isend(..., &req[n])`	`MPI_Irecv(..., &req[n])`
`[Compute]`	`[Compute]`
`MPI_Waitall(..., n+1, req)`	`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) Middleware can't easily nest asynchronous operations on around the MPI API

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 will correct the drawbacks of nonblocking operations as well as put new capabilities into the hands of application developers.

[1] Generalized requests are not a solution.

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 aggregate 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.

[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.

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

```
// Info keys control graph execution behavior
MPIX_Graph_create(&graph, info);
...
// 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>);
...
// Execute operations in graph
MPIX_Graph_execute(graph);
```

Listing 1 – Deferred operations added to graphs

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

```
// Info keys control stream execution behavior
MPIX_Stream_create(&stream, info);
...
// Start stream processing operations that are added
MPIX_Stream_start(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);
...
// Ensure that all operations currently on stream have finished
MPIX_Stream_sync(stream);  // Optional
...
```

Listing 2 – Deferred operations added to streams

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 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" aggregated sets of 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 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.

4.2.1. 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] For full details on graph and stream construction rules, see sections A.3.3 and A.4.2 respectively.
[5] An MPI graph or stream
[6] *You can autopsy or clone it before you bury it, but it's dead.*

4.3. Strong Progress

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

5. Approach

Add description of approach taken, based on A

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 nonblocking operations.

`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.

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

`MPIX_OP_DEF(op, extra_state, req)`

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 short "support" routines, not long computations. The `token` object may be added to a graph or stream.

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

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

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

```
void (*op)(void *extra_state)
```

Possibly other parameters?

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.

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. The `MPI_Info` object may specify key-value pairs to influence the graph's behavior or execution environment.

.....

`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 `MPI_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. `dep_op_id` may also optionally be provided, to create a dependency on an earlier operation already added to the graph.

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_SET_PARAM(graph, name, value)`

INOUT	graph	graph (handle)
IN	name	parameter name (string)
IN	value	pointer to parameter value (choice)

MPIX_GRAPH_SET_PARAM sets the value of a graph parameter variable. 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_GRAPH_EXEC.

. . . . . .

```
MPIX_GRAPH_FREE(graph)
```

INOUT	graph	graph (handle)

MPIX_GRAPH_FREE frees `graph` and sets it to MPI_GRAPH_NULL.

6.3. Memory Operations

TODO: Describe memory operation APIs...

`MPIX_COPY_MEM(dst, src, size)`

INOUT	dst	pointer to beginning of destination memory segment
IN	src	pointer to beginning of sourc memory segment
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.

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

Maybe we want memmove also / instead?

---

7. Use Cases

Use cases from lab notes go here

A. Appendix: An Abstract Data Movement Machine

If there was a chip that provided "assembly" instructions that could be used to implement the API 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.

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)

Comment [QK2]: Anything else that's useful to include here?

Comment [QK3]: 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?

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 1+ source locations to 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, but 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 memory container. 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 container locations are analogous to the existing MPI 'reduce' operation [8] . 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

[8] 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.

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 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.2.3. 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[9]. 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.

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

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. [10]
- *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. [11]

Need better wording here!

A.2.5. Constants

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

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

Comment [QK4]: Interesting variations on barrier could include: a "threshold barrier" operations ("at least 6 of ¡n¿ processes must have reached this barrier before all process may proceed"), a "timeout barrier" ("if all the processes haven't reached the barrier in ¡timeout¿, return an error"), a "remote barrier" ("process A waits until process B reaches the barrier point, process B never waits"), or a "batching barrier" ("processes are batched into sub-communicators of size ¡n¿, as they arrive at the barrier, ideally based on hardware topology")

It might be necessary to provide 'casts' for constants, to specify their datatype.

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".

[10] A pair of synchronous send+receive MPI operations is an example of data synchronization.
[11] The MPI 'barrier' operation is an example of execution synchronization.

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.

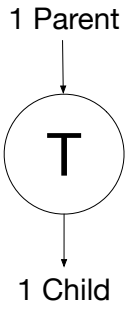

Figure 1 – Task dependencies

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 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. The general form of an unconditional dependency is represented in Fig. 2.

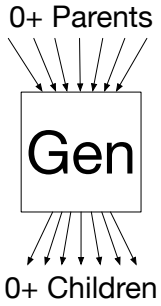

Figure 2 – General form of unconditional organizational graph dependencies

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

- *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.

- *Conditional* dependencies – Conditional 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 switch dependency, which could have weak data dependencies on other concurrently executing operations in the graph that create race conditions when the graph is executed.

Comment [QK5]: Should the threshold be constant or allowed to depend on a graph variable?

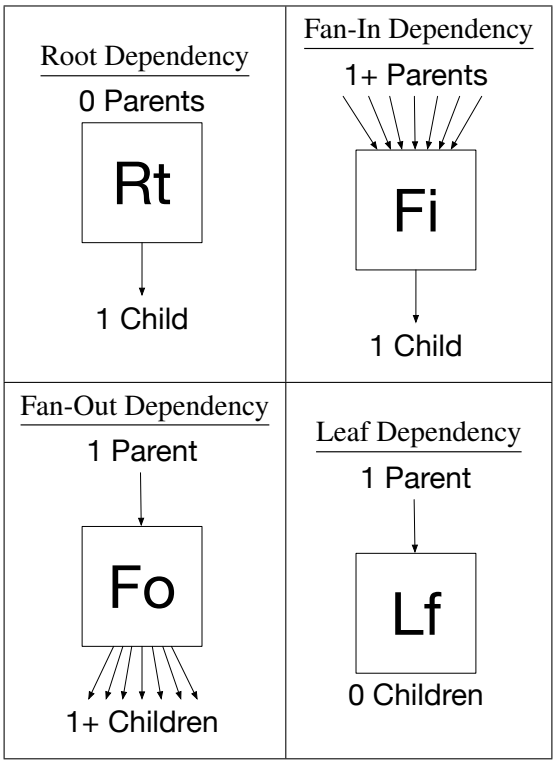

Figure 3 – Common forms of unconditional organizational graph dependencies

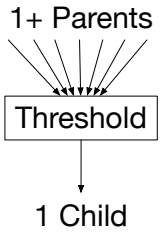

Figure 4 – Threshold organizational graph dependency

Application developers can use other organizational dependencies as preludes to the conditional dependency to eliminate such race conditions.

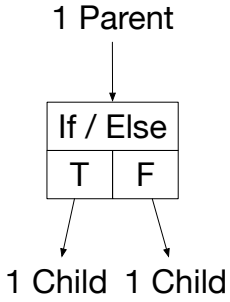

Figure 5 – Conditional organizational graph dependency

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.

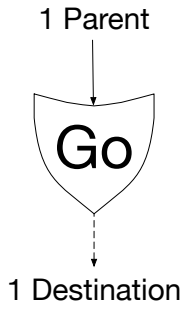

Figure 6 – Control flow operation

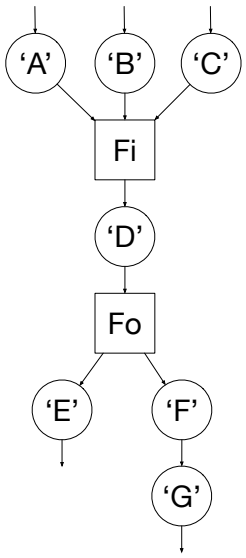

Figure 7 – Example graph for control flow destinations

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.

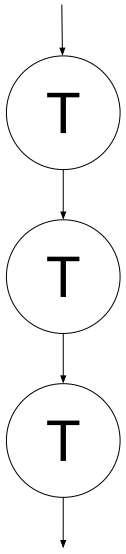

Figure 8 – Connecting multiple task dependencies

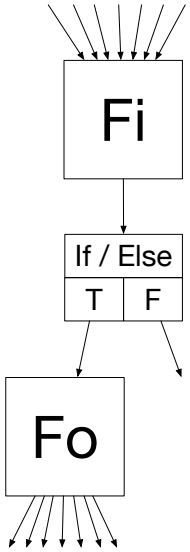

Figure 9 – Connecting multiple organizational dependencies

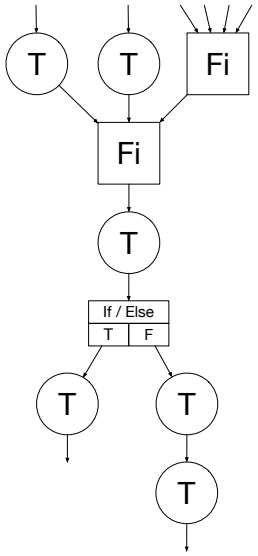

Figure 10 – Combining task and organizational dependencies

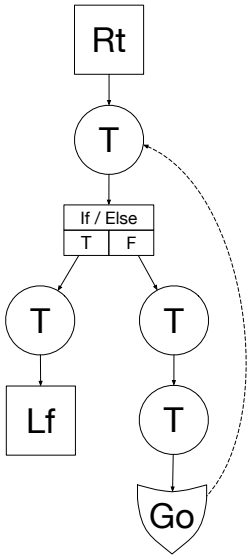

Figure 11 – Combining tasks, organizational dependencies, and control flow operations to create loop in graph

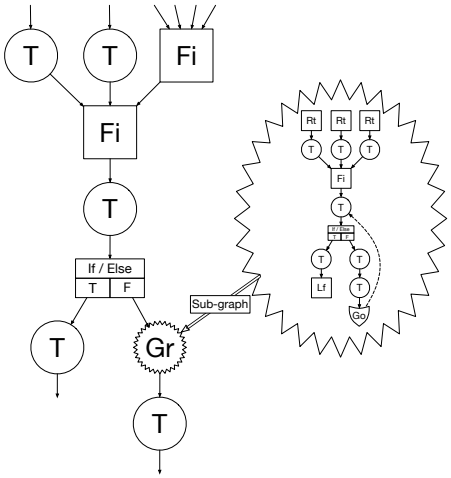

Figure 12 – Combining tasks, organizational dependencies, and sub-graphs

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.

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".

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.

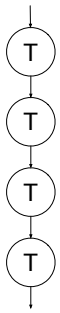

Figure 13 – Tasks in a stream

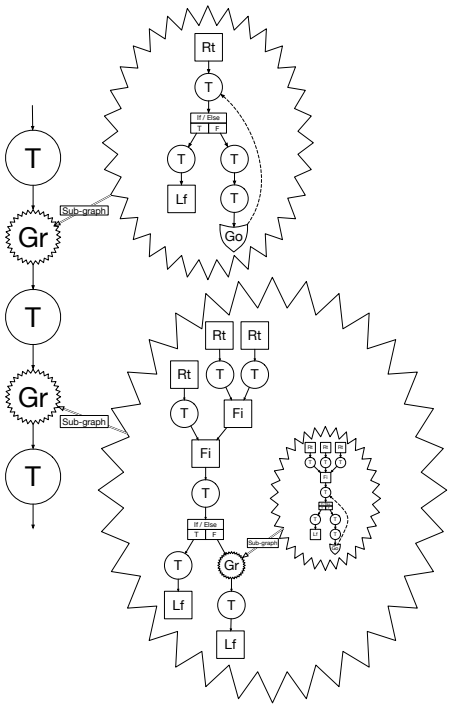

Figure 14 – Tasks and sub-graphs in a stream

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 [QK6]: Not certain it makes sense to have variables for streams.

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 nonblocking operation is invoked when the operation completes. Although the capabilities described in this document do include an asynchronous user callback, they are not comparable operations. A continuation callback requires that the MPI implementation invoke it when the test or wait operation is performed on a request. 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.