Takyon

Version 1.0

User's Guide

Edition 3

abaco SYSTEMS

Introduction

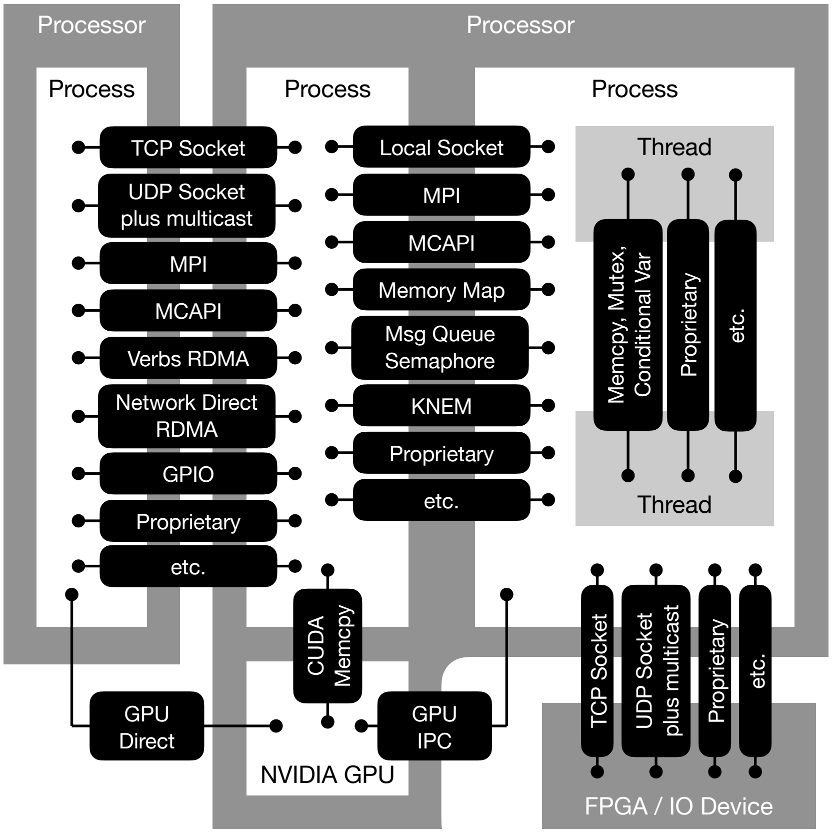
Takyon unifies multiple lower level point to point communication APIs into a simple unified API.

The problem

Many cross-platform communication and signaling standards already exist, but for the most part they are either focused on a particular interconnect hardware, a homogeneous HPC architecture, or locality (inter-thread, inter-process, inter-processor, intra-application).

Each existing standard has different design methodologies, strengths, and weaknesses. Some are very complex requiring hundreds of lines of code just to handle simple concepts. Others intend to be simple, but can get deceptively complex. Some mask important underlying features which can have performance impacts on latency and determinism.

Unfortunately there is no single standard that fits all localities, features, and strengths. The result is high development costs, compromising shortcuts taken, potential to require use of non-de-facto 3rd party proprietary APIs, and confused embedded HPC developers.

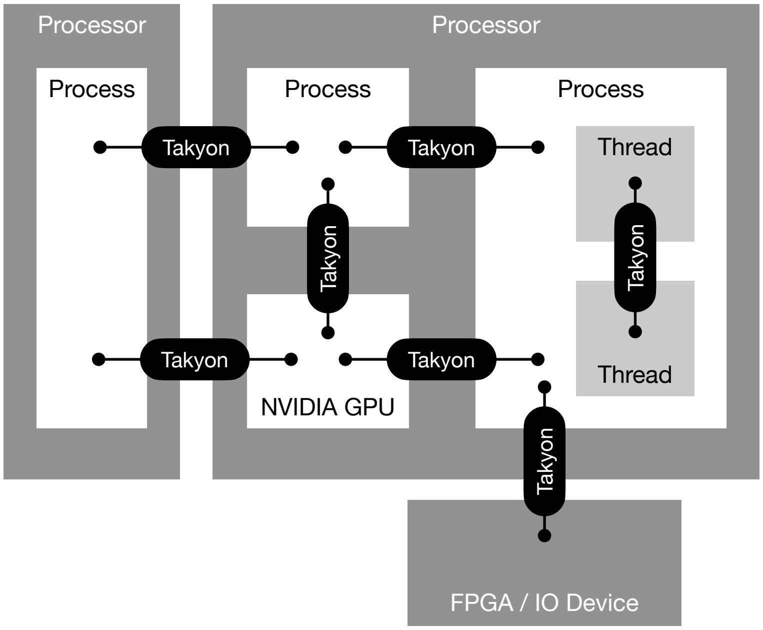


The Solution

At a fundamental level, point to point communication and signaling are about getting data and notifications from one end point to another. This concept is the same for all interconnects and localities.

If most lower level communication and signaling APIs are wrapped into a higher level standard without compromising the lower level features and strengths, then this higher level standard could be a one stop shop for embedded HPC developers.

Takyon is the solution. It's is an efficient layer over many lower level communication and signaling APIs.



Takyon's Prime Directives

The goal is to cater to the embedded HPC engineer who is focused on algorithm development but not communication, and needs the performance and flexibility of lower level communication, and the simplicity of higher level communication concepts.

One Way, Zero Copy, Two Sided

This is the formula to achieving best performance.

- **One Way**: The message is transferred from the sender to the receiver in one underlying communication, without the need for a round trip to ask the receiver where to place the data. Takyon can do this because the application gets the credentials of the receive buffers when the path is created, so there's no need at transfer time to ask the receiver where to put the message.
- **Zero Copy**: The message is transferred without needing to use any intermediate buffering. Just like with 'one-way', since the receive buffers are pre-registered at path creation time, there's no need for intermediate buffering; Takyon can just send the data to the already known receive buffer locations.
- **Two Sided**: Integrated into the message transfer is an implicit notification to the receiver when the message has arrived and is ready to be processed, avoiding any expensive explicit secondary signaling message or method (like polling on a receive buffer memory location) to do the notification. Note that even for unconnected communications, like multicast, there are still two sides involved, and there is still an implicit signaling method to coordinate when data has arrived at the receiver.

Minimal Implicit Synchronization

Synchronizing requires messaging from one end point to the other and can perturb determinism and latency. The only implicit synchronization that Takyon uses is to notify the sender when the message has left and notify the receiver when the message has arrived. All other synchronization is left to the application, and should be used when:

- The receiver is ready for more data and the sender can start another message transfer.
- The sender's buffer can be filled (in the case where the sender and receiver are sharing a memory buffer).

More details on this is explained in the next chapter.

Application Level Fault Tolerance

This is the formula to error recovery. This means that if something goes wrong with a communication path, it can be detected and either re-established or use an alternative path or method to make sure the application continues reliably.

- **Dynamic Connections**: Takyon paths can be created and destroyed at any time during the application's life cycle without effecting any other established communication paths. Multiple paths can be created between the same two endpoints using the same or different interconnects.
- **Timeouts**: When sending and/or receiving, there may be some amount of time that passes where the Takyon path is no longer considered responsive. If a Takyon path reports a timeout, then the application can take the appropriate action to keep running reliably; e.g. use an alternate communication path.
- **Disconnect Detection**: While timeouts can imply degraded communication paths, most modern interconnects can also detect when a path has disconnected due to some failure. If a Takyon path reports a disconnection, then the application can take the appropriate action to keep running reliably.

Follow the Intuition, Not the Hardware

Fundamentally, communication is about getting data from A to B. When using a higher level communication API, there should be no need to have different function calls for each type of interconnect or for different localities (thread, process, processor). Takyon's API is based on the following five communication concepts, and therefore only has five core functions:

- **Create**: create a communication path to a remote endpoint
- **Send**: send a message to the remote endpoint (blocking and non-blocking)
- **Non-Blocking Send Finished Test**: check if a previously started non-blocking send is complete
- **Receive**: receive a message
- **Destroy**: destroy the path

Point to Point Communication 101

This chapter will cover the fundamentals of point to point communication and how it was designed into Takyon to keep it simple and maintain best possible performance without any significant tradeoffs.

Note: In this chapter, pseudo code will be used to describe the concepts instead of actual Takyon functions.

Paths and Endpoints

A path is just a concept that logically describes the connectivity between two endpoints. When data is sent from the sender's endpoint, it's pushed down a logical path intended to get to the receiver's endpoint.

Connected (reliable) versus Connectionless (unreliable, multicast)

A path can be connected or connectionless.

A connected path knows about the remote endpoint. One cannot exist without the other, so if one of the endpoints fail, then this will force the remote endpoint to be invalid. Connected paths are bi-directional; i.e. either endpoint can send data to the remote endpoint. This is the type of connection to use when data transfers need to be reliable and ordered.

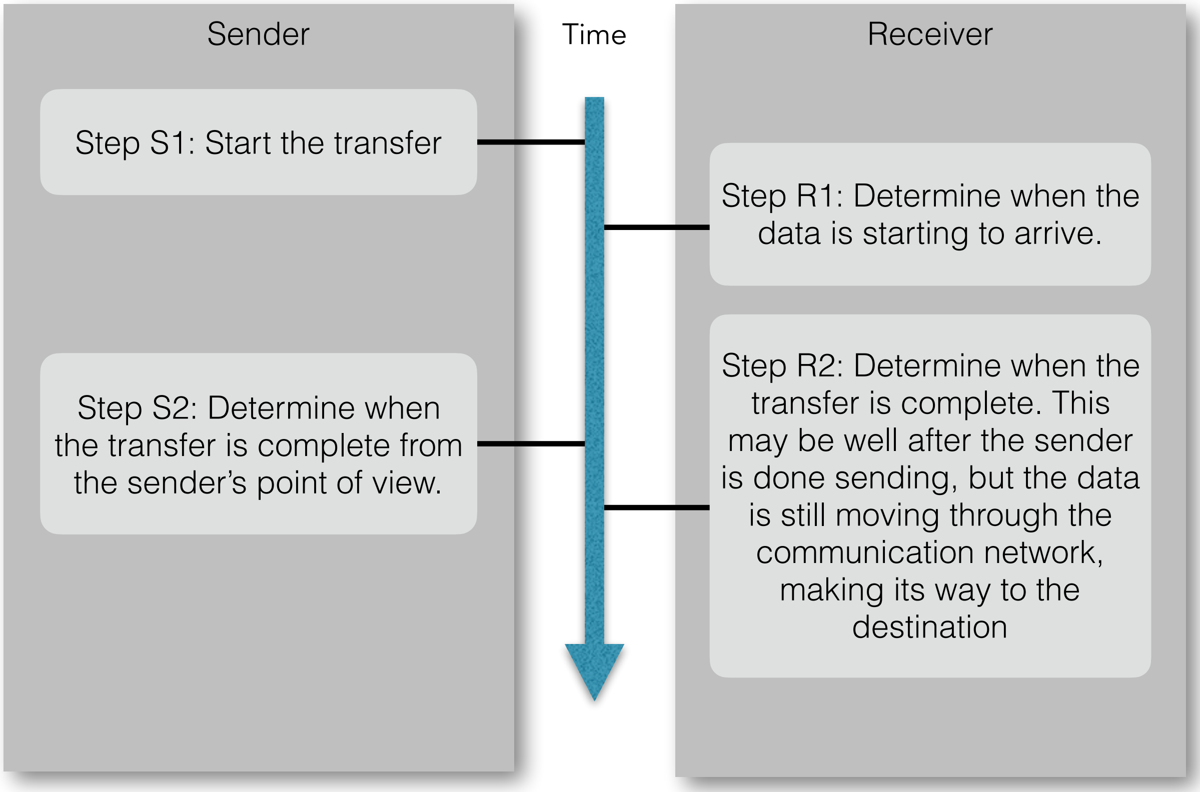
A connectionless path is one sided, and does not require the remote endpoint to exist. If data is sent, and the other side does not exists, the data will just be dropped on the transport. If both endpoints do exists, the receiver is not guaranteed to get all the messages the sender sends, and is not guaranteed to get the messages in the order sent. Connectionless also allows for multicasting where one sender can send to multiple receivers. Connectionless communication is useful for things like live video streaming, live audio streaming, analog to digital signal streaming, and digital to analog signal streaming, where it's OK to occasionally lose data or get data in a different order than the order sent. The data size transferred with connectionless paths is usually limited, typically to around 64 Kbytes (the typical size of a UDP datagram) or less (sometimes the MTU size).

Messages

Takyon is designed to pass messages: i.e. what is sent by a single call to send() is the same as what is received by a single call to recv(). With connectionless, a message could be dropped, but if it is received, it will be the entire message.

Data Transferring

Intuitively communication is about moving data from endpoint A to endpoint B or visa versa. To break it down a little further, there's two phases of sending and two phases of receiving.


Sender
Time
Receiver
Step S1: Start the transfer
Step R1: Determine when the data is starting to arrive.
Step S2: Determine when the transfer is complete from the sender's point of view.
Step R2: Determine when the transfer is complete. This may be well after the sender is done sending, but the data is still moving through the communication network, making its way to the destination

That's all there is to it.

Note: that S1, S2, R1, and R2 (from the diagram above) will be referenced throughout this chapter. Also, the following pseudo code will also be referenced:

- sendStart() == S1
- sendIsFinished() == S2
- send() == S1 + S2
- recv() == R1 + R2

Transport Memory

Many communication packages define/register the source and destination transport memory at the time of the transfer.

Sender

```
send(data, bytes)
```

Receiver

```
recv(data, &bytes_received)
```

Important: One of the most fundamental aspects of Takyon, which is very different from many communication packages, is that Takyon requires the sender and receiver transport memory to be defined and registered when the communication path is established. This is a Takyon buffer. This means the Takyon send and receive functions will not need a data address passed in as an argument since they are already known.

Sender

```
send(bytes)
```

Receiver

```
recv(&bytes_received)
```

This may seem odd at first, but the improved performance is potentially a big benefit.

Takyon also allows a source and destination offset (in bytes) to be specified when sending, so the application can start from any byte in the Takyon buffer. If managed correctly, a single area of contiguous memory can be used by multiple Takyon buffers even across multiple Takyon paths. This is helpful for collective functions like scattering and gathering.

Blocking Transfers

The easiest form of transferring is to do steps S1 and S2 together in a single send function, and do steps R1 and R2 together in a single receive function.

Sender

```
send(bytes)
```

Receiver

```
recv(&bytes_received)
```

This is the common use case of transferring because it's simple to understand.

Non Blocking Transfers

This allows the sender to do step S1 and S2 as separate steps with intermediate processing between the steps.

Sender

```
sendStart(bytes)
// do something useful here
sendIsFinished()
```

Receiver

```
recv(&bytes_received)
```

Non blocking transfers are useful if the interconnect allows background transfers (no CPU interaction is needed while the data is being sent). This allows extra processing to occur at the same time the transfer is in progress.

The receiver could also be broken up into two separate functions, but it's typically not useful to know when data is starting to arrive, but instead only when it has completely arrived. Therefore Takyon only has one receive function.

Application Controlled Synchronization

Important: Another characteristic of Takyon achieving best performance, is the transfer needs to be one way. This means that the sender does not implicitly know when the receiver is ready for more data. This puts the responsibility of the synchronization on the application, but **this is a good responsibility**, because it allows the application to choose the best time to do the synchronization, and not waste time doing un-needed synchronization. Some applications are self synchronizing due to round trip communications. If the application does not naturally have round trip synchronization, then it's easy to use Takyon to do a zero byte transfer to explicitly do the synchronization.

INCORRECT METHOD (**BAD**): The following represents sending without synchronization:

Sender

```
while (1) {
  prepareData(data, bytes)
  send(bytes)
}
```

Receiver

```
while (1) {
  recv(&bytes)
  processData(data, bytes)
}
```

In this case the receiver's data could be overwritten while in the middle of processing, or worse, cause a crash because the interconnect is not designed to handle this.

CORRECT METHOD (**GOOD**): The proper way to guarantee correctness for all interconnects (even shared memory interconnects), is to add explicit synchronization.

Sender

```
while (1) {
  prepareData(data, bytes)
  send(bytes)
  recv(NULL) // Wait here
}
```

Receiver

```
while (1) {
  recv(&bytes)
  processData(data, bytes)
  send(0)  // 0 byte message
}
```

The sender will block after sending a message. The receiver sends the synchronization signal after processing the data.

BETTER METHOD: The above is correct for the general case, but if the interconnect is not using shared memory (one buffer shared by sender and receiver) then it can be improved. It would be better to block before sending instead of after sending, allowing data to be collected/generated on the sender without waiting for the receiver to finish processing the previous data.

Sender

```
while (1) {
  prepareData(data, bytes)
  recv(NULL) // Wait here
  send(bytes)
}
```

Receiver

```
while (1) {
  send(0)  // 0 byte message
  recv(&bytes)
  processData(data, bytes)
}
```

This is more efficient since the sender and receiver will be processing concurrently.

If the application is using multiple transfer buffers, synchronization might only be needed after all of the buffers have been used up. More on this in the "Multi Buffering" section below.

As you can see, there are many choices of when to use synchronization. This shows why it may be detrimental if Takyon used implicit synchronization with every transfer; most of them may be unnecessary. This would reduce performance, and perturb determinism.

Shared Memory

Some Takyon interconnects support shared memory. This is where the sender and receiver actually point to the same memory locations. When the send() is called, it only needs to send a signal to the receiver no matter how many bytes the message is. It's really just sharing a pointer to a memory address.

Synchronization with Shared Memory

If the interconnect is defined to use shared memory, then that means the sender and receiver are actually pointing to the same buffer (they don't have their own independent buffers). If this is the case, the application will need to do more sophisticated synchronization to know when it is 'safe' to start filling the send side buffer. I.e. it is 'safe' when the receiver is not currently processing memory on the same buffer that the sender is about to fill.

Multi Buffering

Double and triple buffering is a common practice to overlap processing and transfers. A well developed multi-buffered application will minimize idle time, achieving great overall throughput and CPU utilization.

Takyon builds multi-buffering into the core APIs, making it easy for application developers to use multi-buffering in the application. Buffers are independent of each other, so when a transfer is busy on one buffer, the sender side can start filling data on another buffer and the receiver side can process the previous message it received. This is the idea of keeping the Takyon path and the processing cores on both endpoints busy.

Here's an example of multi buffering and how synchronization is used (if not using a shared memory interconnect):

Sender

```
nbufs = 3
buf = 0
while (1) {
  prepareData(data[buf], bytes)
  if (buf==0) recv(buf, NULL)
  send(buf, bytes)
  buf = (buf+1) % nbufs
}
```

Receiver

```
nbufs = 3
buf = 0
while (1) {
  if (buf==0) send(buf, 0)
  recv(buf, &bytes)
  processData(data[buf], bytes)
  but = (buf+1) % nbufs
}
```

In this example, synchronization is only done once for all the buffers. This minimizes the overhead for synchronization without compromising correctness.

This example is a great way to show that if Takyon had implicit synchronization for each transfer, it would degrade performance. This is why Takyon does not do any extra implicit synchronization and leaves it to the application.

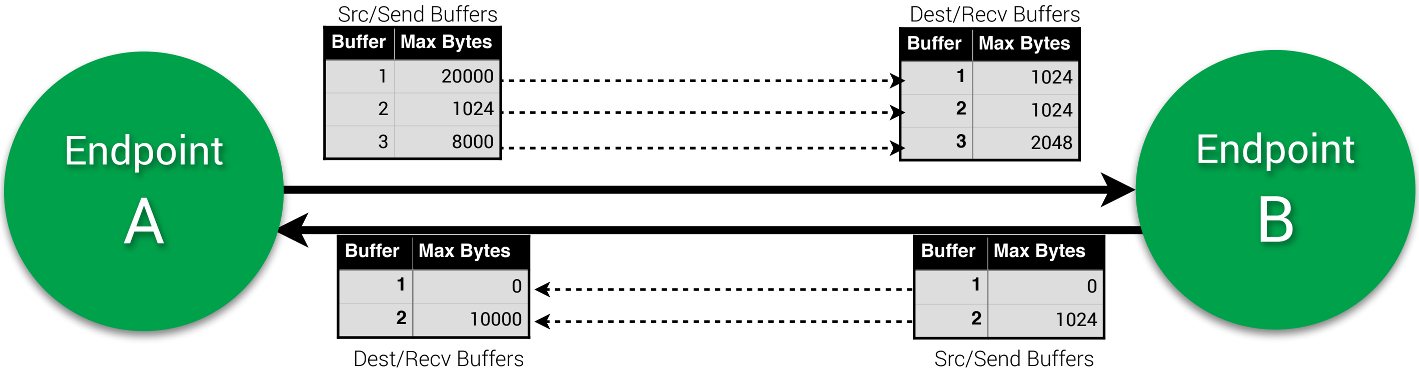
Polling and Event Driven

Takyon performance is also defined by the method of checking for transfer progress. When waiting for a call to send() or receive() to complete, there are only two ways to do this:

- **Polling**: This uses CPU-based looping to keep checking if the transfer is complete. There are no context switches, so this is very responsive to the exact moment the transfer is complete. The drawback is that the compute core will not be available to do any other processing while Takyon is constantly checking to see if the transfer is complete. Plus a side effect is that the compute core will draw extra power and increase its temperature.
- **Event Driven**: This uses an OS-based mechanism to put the thread to sleep while waiting for the underlying communication interconnect to complete the transfer. While the thread is sleeping, the compute core can be busy doing other work. When the transfer is complete, a signal will be sent to the OS to let it know it can wake up the sleeping thread. Knowing when the transfer completed will not be as responsive as polling due to the context switches, but it will free up the compute core to allow other threads and processes to use the core while the communication transfer is in progress.

Bi-Directional Communication and Buffer Sizes

Once a connected Takyon path is established, transfers can be done in either direction. This allows the number of created paths to be minimized. Since the goal is to pre-register memory when the path is created, each side of the path must know the largest message size it can transfer. A decision may be to allow one direction to send large data blocks, and the other direction have no message space at all but be able to send zero byte messages to synchronize transfers. Another decision is to have the same buffer size for both directions.



Src/Send Buffers

Buffer	Max Bytes
1	20000
2	1024
3	8000

Dest/Recv Buffers

Buffer	Max Bytes
1	1024
2	1024
3	2048

Buffer	Max Bytes
1	0
2	10000

Dest/Recv Buffers

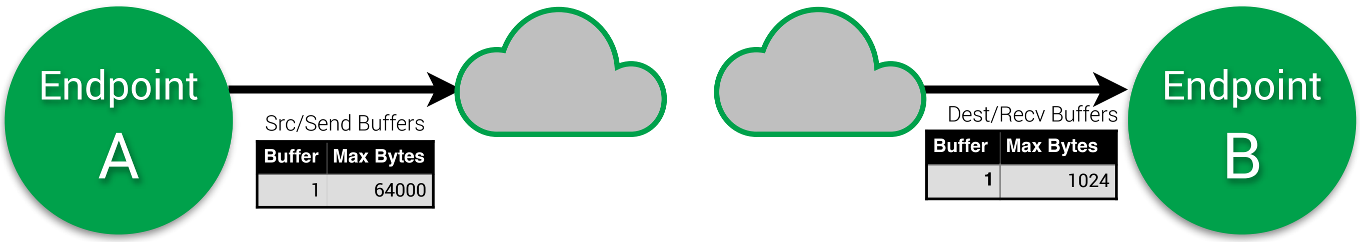
Buffer	Max Bytes
1	0
2	1024

Src/Send Buffers

NOTE: This may be unintuitive, but the send/recv buffer pairs do not need to be the same size. This is useful for collective communication like scatter or gather, where the one side of the communication may have a lot more contiguous data than the other side (see Takyon's "scatter_gather" example).

Uni-Directional Communication and Buffer Sizes

Once a connectionless Takyon path is established, transfers can only be done in one direction. Furthermore, the max buffer size may be limited to 64 Kbytes (the typical size of a UDP datagram) or less (the MTU size). This is usually good for things like live streaming.



Src/Send Buffers

Buffer	Max Bytes
1	64000

Dest/Recv Buffers

Buffer	Max Bytes
1	1024

NOTE: This may be unintuitive, but Takyon can still support multiple buffers on the sender or receiver. This may be useful for organizing the message in some helpful way. The oddness is the remote side will not have any knowledge of how many buffers the other side has, nor the order of the messages arriving on the receiver. (see Takyon's "connectionless" example).

Locality & Interconnects

Takyon is designed to work between any two endpoints (localities).

- Thread to thread in the same process
- Process to process on the same OS
- Process to process between different CPUs
- Application to application

And to work with most any interconnect: threads, shared memory maps, sockets, RDMA, IO devices, etc.

This makes Takyon a truly portable communication package. When a path is created, the application can choose which underlying interconnect is used base on the locality of the end points. In many cases, there will be more than one interconnect to choose from. This gives the application the control to create multiple paths to use different interconnects in different ways.

IO Device Communication

IO Devices are typically one sided (connectionless). An application would typically receive from an input device (but not send to it), and send to an output device (but not receive from it). For example, this could facilitate communication to and from an FPGA, Lidar, Camera, etc. Takyon does not look at IO devices differently than any other typical point to point connectionless communication.

Shared Memory, IO Device Memory, GPU Memory

Takyon buffers are not restricted to CPU ram, but can also use shared, IO device, or GPU memory (e.g. CUDA). It's up to the Takyon implementer to add the support for any particular interconnect.

Scalability & Dataflow

Communication should be scalable for any size system. It does mean there needs to be a discipline for how to design the scalability. Developers need to spend time mapping an application to a system, with a specific number of processors, cores, interconnects, memory sizes, IO devices, etc. If the developer hardcodes the application to a particular system, then it will make it difficult to migrate to a future system with a different number of resources. Designing an application dataflow could be done at a logical level regardless of the final deployed system. In this way the application can be tested with minimal resources. When dataflow is scaled out, a well designed application should not need any source code modification.

If you have an extremely large system, hundreds of processors, then it would probably not be feasible to have direct communication connections between all processor pairs. For example, if you have a 2D grid of processors, each processor may only need to connect to adjacent processors. If the application needs to send a message to a far away processor, it could use multiple hops to make it to its destination.

Scalability should not be enforced by a communication package, but instead should be enforced by good dataflow design. See the Takyon examples that use the Takyon graph file description: hello_world_graph, barrier, reduce, pipeline, scatter_gather.

Strided Messages

Very few modern interconnects support strided transfers. Even common open-standard interconnects such as Sockets and OFED RDMA don't support strided messages.

A common example of strided data is an array of complex data (pair of real and imaginary values). It may be required by the application to send the real values to one endpoint, and the imaginary values to a different endpoint.

If a communication API supports strided, but the underlying hardware can't, this will result in a significant compromise (mysterious bad performance) with two options (hidden away in the communication API):

- Memory is dynamically allocated to reorganize the data into separate contiguous blocks and then sent down separate paths.
- Each strided block of the message needs to be sent separately; e.g. if each block is 4 bytes, but the message is megabytes, then this results in millions of tiny transfers.

When dealing with strided data, it should be the responsibility of the application, not the communication API, to efficiently handle strided data. Therefor Takyon does not currently support strided transfers.

Where Are the Callbacks?

The concept of a callback is that the application waits for a transfer completion through a registered function that gets called by an event handler in a different thread from the one that started the transfer. Callbacks may seem like they fit well with communication, but if the application does not have an event loop that collects all events, callbacks will be confusing and difficult to manage in a way that is thread-safe for communication. Unlike graphical user interfaces (GUIs) that require an event loop, most lower level communication APIs don't have an event loop concept (i.e. callback functions to notify when a transfer is complete).

A communication API could require an event loop, forcing all communication into a single thread. This would essentially serialize all communication events (send done & recv done events) and may degrade application performance due to the extra context switches.

An alternative could be to have a separate background thread for each created path. A callback would be called from this background thread. But now this requires the application to coordinate between threads and this also adds extra context switches. This is complex enough that it would likely be a feature that is rarely used or not understood correctly. Even if well understood, the extra performance hit may be undesirable.

Due to these complexities for the application developer, Takyon does not have a built-in callback method. That said, this does not preclude the application developer from implementing one of the two callback methods mentioned above. It would just need to be done explicitly by the application instead of being implicit in the Takyon API.

Open Source Extensions; including Collective Communication

Takyon includes some helpful extensions for common functionality in many applications using communication. This includes functions for:

- Current time, sleeping
- Endian testing, endian swapping
- Shared memory management
- Collective communication: barrier, reduce, scatter, gather, and generic layouts. This allows communication to be written in a similar way to MPI's collective functions.
- Graph file to describe scalable dataflow

Theses functions are provided as open source code to make it easy to modify as needed. See the 'Takyon/extensions/takyon_extensions.h' header file for the set of available functions. The source code is in the same folder.

Many of the Takyon examples make use of the open source extensions.

Core API Reference

takyonCreate

```
    TakyonPath *takyonCreate(TakyonPathAttributes *attributes)
```

Then create the Takyon path's endpoint. If the other endpoint is also a Takyon endpoint, then takyonCreate() will also need to be called for that endpoint.

For connected interconnects, this creates a reliable communication path between the two endpoints. Messages will arrive in the same order as they are sent, also message won't be dropped.

For connectionless interconnects, the other endpoint does not need to exists for this endpoint to be created. Also, messages may be dropped before making it to the receiver, and order is not guaranteed.

The properties of the path are defined by the attributes, and must be set at time of creation. They cannot be changed once the path is created. If different behavior is needed over time, then create multiple paths to handle the different behaviors.

The attributes are passed in from a pointer to a TakyonPathAttributes structure (allocated by the application before this call is made), which contains the following fields:

Field	Description
char interconnect[TAKYON_MAX_INTERCONNECT_CHARS]	This defines the interconnect to use for this path, a unique identifier to distinguish this path from all other paths using the same interconnect, and any optional parameters supported by the interconnect. See the implementation's release notes for a list of supported interconnect names, and the set of supported parameters. For connected interconnects, both end points must use the same interconnect name but the interconnect parameters on each endpoint are implementation specific. The size, in bytes, of the *interconnect* field is TAKYON_MAX_INTERCONNECT_CHARS, which needs to include the null terminating character of the string.
bool is_endpointA	One of the endpoints must be designated as endpoint A by setting this field to 'true'. If set to 'false', then it's endpoint B. For connected interconnects, there is no functional difference after the connection is made, and either end point can do transfers.
bool is_polling	This sets if the communication path is event driven or polling when determining if a transfer is complete. Set to 'true' to use polling, or set to 'false' to use event driven. Some interconnects may require both end points to have the same value.
bool abort_on_failure	This determines if the application should abort if a failure occurs. • Set to 'true' if the application will call abort() if a failure is detected. Before aborting, a helpful error message will be printed to stderr. Using this mode allows the application to avoid the need for error checking the return values of the Takyon functions, resulting in cleaner source code. • If set to 'false', then the Takyon functions will return even if a failure occurs. All returned values need to be checked to have a reliable application.

Field	Description
uint64_t verbosity	There may be a need to see what’s going on within the Takyon functions. This will allow the application to print different types of information to stdout/stderr. If set to 0, the takyon functions do not print any information. Otherwise, do a bitwise or-ing of the following values: • TAKYON_VERBOSITY_NONE - A convenience to show there is no verbosity, this value is 0, but if masked with other values, then this will be ignored. • TAKYON_VERBOSITY_ERRORS - Print (to stderr) details of what caused an error. If this is not used then errors will not get printed out unless 'abort_on_failure' is set to 'true'. • TAKYON_VERBOSITY_CREATE_DESTROY - Print (to stdout) basic high level information during the create and destroy functions. • TAKYON_VERBOSITY_CREATE_DESTROY_MORE - Print (to stdout) extra details during the create and destroy functions. • TAKYON_VERBOSITY_SEND_RECV - Print (to stdout) basic high level details during the send and receive functions. • TAKYON_VERBOSITY_SEND_RECV_MORE - Print (to stdout) extra details during the send and receive functions.
double path_create_timeout	The timeout period, in seconds, to wait for takyonCreate() to complete creating the endpoint. If the endpoint can't be created within the timeout period, then the create function returns NULL.
double send_start_timeout	The timeout period, in seconds, to wait for takyonSend() to start transfer of the message. Some interconnects are guaranteed to get started without ever waiting and will ignore this timeout, but some interconnects may have to wait for the resource to become available. If takyonSend() times out using this timeout, this is not an error, and sending can be attempted again later.
double send_finish_timeout	The timeout period, in seconds, to wait for a send, that already started transferring the message, to complete. If the send is blocking, then it will be used by takyonSend(). If the send is non blocking then this timeout will be used by takyonSendIsFinished(). If this timeout occurs while in takyonSend(), this is considered an error, and the path must be destroyed. If the send is blocking, then this timeout should be greater than zero to provide a reasonable amount of time for the already started transfer to complete. Only interconnects that can detect partial transfers will support this timeout.
double recv_start_timeout	The timeout period, in seconds, to wait for takyonRecv() to start receiving a message. If takyonRecv() times out using this timeout, this is not an error since no message has started arriving, and receiving can be attempted again later.
double recv_finish_timeout	The timeout period, in seconds, to wait for takyonRecv() to receive a complete message that is in the process of being received. If the timeout occurs, then this is considered an error, and the path must be destroyed. This timeout should be greater than zero to provide a reasonable amount of time for the already started transfer to complete.
double path_destroy_timeout	The timeout period, in seconds, to wait for takyonDestroy() to gracefully destroy the endpoint, which may include a coordination with the remote endpoint for connected interconnects. If the connection cannot be gracefully shutdown in the timeout period, then the connection is forcibly destroyed and this is considered an error, and in this case takyonDestroy() will return an error string instead of NULL.

Field	Description
TakyonCompletionMethod send_completion_method	Determines if takyonSend() is blocking or non blocking. Select one of the following values: • TAKYON_BLOCKING - Wait for takyonSend() to complete the send. • TAKYON_USE_IS_SEND_FINISHED - Start the transfer using takyonSend(), then use takyonIsSendFinished() to know when the send is complete. Interconnects that don't support non-blocking will complete the transfer before takyonSend() returns, but takyonSendTest() will still need to be called to complete the transaction.
TakyonCompletionMethod recv_completion_method	Currently, this must be set to TAKYON_BLOCKING, which means takyonRecv() will block waiting for the transfer to complete. To avoid blocking if no message arrives, use a 'recv_start_timeout' of zero or greater.
int nbufs_AtoB	Defines the number of transfer buffers from A (the source) to B (the destination). One of nbufs_AtoB or nbufs_BtoA must be greater than zero. If both endpoints are Takyon, then some interconnects will require both endpoints to have the same value for nbufs_AtoB.
int nbufs_BtoA	Defines the number of transfer buffers from B (the source) to A (the destination). One of nbufs_AtoB or nbufs_BtoA must be greater than zero. If both endpoints are Takyon, then some interconnects will require both endpoints to have the same value for nbufs_AtoB.
uint64_t *sender_max_bytes_list	This is a pointer to the endpoint's list of send buffer byte sizes where the size of the list must be *nbufs_AtoB* items if it's endpoint A or *nbufs_BtoA* items if it's endpoint B. The sender and receiver can have different sizes for the same buffer. An item in the list can be zero, allowing for zero byte messages to be sent.
uint64_t *recver_max_bytes_list	This is a pointer to the endpoint's list of receive buffer byte sizes where the size of the list must be *nbufs_BtoA* items if it's endpoint A or *nbufs_AtoB* items if it's endpoint B. The sender and receiver can have different sizes for the same buffer. An item in the list can be zero, allowing for zero byte messages to be sent.
size_t *sender_addr_list	This is a pointer to a list of memory addresses where the size of the list must be *nbufs_AtoB* if this is endpoint A or *nbufs_BtoA* if this is endpoint B. If an item in the list is set to 0, then Takyon will allocate the appropriate amount of memory aligned to a page boundary. If the corresponding byte size is 0, then no memory is allocated, but this will still allow Takyon to do zero byte transfers. If memory was allocated by Takyon, it will automatically free this memory when the path is destroyed. If an item in the list is not 0, then it must be a valid memory address (casted to a *size_t*) pointing to the appropriate memory address containing the number of bytes defined by *sender_max_bytes_list*. Some interconnects may have additional restrictions or requirements. Read the implementation notes to know the details.

Field	Description
size_t *recver_addr_list	This is a pointer to a list of memory addresses where the size of the list must be *nbufs_BtoA* if this is endpoint A or *nbufs_AtoB* if this is endpoint B. If an item in the list is set to 0, then Takyon will allocate the appropriate amount of memory aligned to a page boundary. If the corresponding byte size is 0, then no memory is allocated, but this will still allow Takyon to do zero byte transfers. If memory was allocated by Takyon, it will automatically free this memory when the path is destroyed. If an item in the list is not 0, then it must be a valid memory address (casted to a *size_t*) pointing to the appropriate memory address containing the number of bytes defined by *recver_max_bytes_list*. Some interconnects may have additional restrictions or requirements. Read the implementation notes to know the restrictions.
char *error_message	This fields is used to the report errors in a human readable way, which includes the call stack with file and line numbers within the Takyon core source code. This field should not be set by the application, because takyonCreate will allocate the memory for this field. This field is used to report errors if they occur with takyonCreate(), takyonSend(), takyonIsSendFinished(), or takyonRecv(). If an error occurs with takyonDestroy() the string is returned directly and must be freed by the application. Also, if the error occurs with takyonCreate(), then the attributes.error_message field must be freed if it's not NULL. In all other cases, this memory should not be freed.

If the path was successfully created, then a pointer to a TakyonPath structure is returned. This structure is opaque except for the *path->attrs* field. The attrs field allows the path to know all of it properties, including any memory buffers that Takyon allocated, which are located in the lists *sender_addr_list[]* and *recver_addr_list[]*.

If the path was not created due to an error or was not created within the timeout period *attributes->path_create_timeout*, then one of the following will occur:

- If *attributes->abort_on_failure* is *false*, then NULL is returned, and an error message is stored in *attributes->error_message* if this field is not NULL, which means the application must free this field.
- If *attributes->abort_on_failure* is *true*, then an error message is printed to stderr, and then the process will call abort().

takyonSend

```
bool takyonSend(TakyonPath *path, int buffer_index, uint64_t bytes, uint64_t src_offset, uint64_t
dest_offset, bool *timed_out_ret)
```

Starts a message transfer.

The transfer will be blocking if the path was created using TAKYON_BLOCKING for *send_completion_method*.

If the path was created using TAKYON_USE_IS_SEND_FINISHED for *send_completion_method* then *takyonIsSendFinished*() will need to be called to complete the transfer, even if the underlying interconnect does not support non-blocking.

NOTE: Even if the sender completes the transfer, the receiver may still not have all the data, as some of it may still be in the network on its way to the receiver.

buffer_index (starting at 0) represents which memory buffer address in *path->attrs.sender_addr_list[]* that will be used as the source of the transfer.

bytes is the number of bytes that will be transferred. This must be between 0 and the max size of the buffer bytes.

src_offset is the offset from the start of the source's memory block where the transfer will start from. This must be between 0 and the max size of the buffer.

dest_offset is the offset from the start of the destination's memory block where the transferred data will start from. This must be between 0 and the max size of the buffer.

timed_out_ret is a pointer to a bool (supplied by the application) or NULL if *path->attrs.send_start_timeout* and *path->attrs.send_finish_timeout* are both set to TAKYON_WAIT_FOREVER. If not NULL, then after the call completes, then the bool variable will be set to true if the transfer timed out, otherwise it will be false.

If the function succeeds, then it will return true, and **timed_out_ret,* if not NULL*,* will be set to false.

If the function times out before the transfer has started, then it will return true, and **timed_out_ret* will be set to true. In this case, this call was a no-op; the communication path has not yet done anything, so it's still a valid path, and this call can be made again on the same buffer.

If the function fails, then the connection is in a bad state, and one of the following will occur:

- If *path->attributes.abort_on_failure* is *false*, then *false* is returned, and an error message is stored in *path->attributes.error_message*. The path is in a bad state and needs to be destroyed with *takyonDestroy*()
- If *path->attributes.abort_on_failure* is *true*, then an error message is printed to stderr, and then the process will call abort().

takyonSendTest

```
bool takyonSendTest(TakyonPath *path, int buffer_index, bool *timed_out_ret)
```

This blocks until a previously started non blocking send complete's the transfer.

This function should only be called if the path was set up using TAKYON_USE_IS_SEND_FINISHED for *send_completion_method*.

buffer_index needs to match the buffer index set in the *takyonSend*() call that started the transfer.

timed_out_ret is a pointer to a bool (supplied by the application) or NULL if *path->attrs.send_finish_timeout* is set to TAKYON_WAIT_FOREVER. If not NULL, then after the call completes, then the bool variable will be set to true if the transfer timed out, otherwise it will be false.

If the function succeeds, then it will return true, and **timed_out_ret,* if not NULL, will be set to false.

If the function times out, then it will return true, and **timed_out_ret* will be set to true. This indicates that the started transfer is not yet complete and this call can be made again later to see if it's complete then. The application may also decide that too much time elapsed and that the path is no longer responsive enough to be considered valid, and can take the appropriate action (application defined fault tolerance).

If the function fails, then the connection is in a bad state, and one of the following will occur:

- If *path->attributes.abort_on_failure* is *false*, then *false* is returned, and an error message is stored in *path->attributes.error_message*. The path is in a bad state and needs to be destroyed with *takyonDestroy*()
- If *path->attributes.abort_on_failure* is *true*, then an error message is printed to stderr, and then the process will call abort().

takyonRecv

```
bool takyonRecv(TakyonPath *path, int buffer_index, uint64_t *bytes_ret, uint64_t *offset_ret, bool
*timed_out_ret)
```

This blocks until the message arrives from the remote endpoint.

This call does not need to occur before the remote endpoint starts sending. This is because the sender already has a handle to the destination memory, and in general can complete the transfer without any explicit interaction from the receiver. If the data has already arrived, then this call will not block at all.

NOTE: With connected stream interconnects, like sockets, the takyonSend() may block until this function is called because a send-side socket needs to coordinate with the recv-side socket to complete large transfers.

buffer_index represents which memory buffer address in *path->attrs.recver_addr_list[]* that will be used to hold the received data. For connected interconnects, this must match the buffer index set by the call to takyonSend().

bytes_ret is the number of bytes that was received if the transfer completes successfully. If the send side set bytes to 0, then this will also be zero, which likely means it was just used for synchronization/notification. This can be set to NULL if the receiver already knows how many bytes it will receive.

offset_ret is the offset of the received data from the start of the destination's memory buffer address. This can be set to NULL if the receiver already knows the offset.

timed_out_ret is a pointer to a bool (supplied by the application) or NULL if *path->attrs.recv_start_timeout* and *path->attrs.recv_finish_timeout* are set to TAKYON_WAIT_FOREVER. If not NULL, then after the call completes, then the bool variable will be set to true if the transfer timed out, otherwise it will be false.

If the function succeeds, then it will return true, and **timed_out_ret,* if not NULL, will be set to false.

If the function times out before a message starts to arrive, then it will return true, and *timed_out_ret will be set to true. This can be considered a no-op, and this call can be made again later on the same buffer index. The application may also decide that too much time elapsed and that the path is no longer responsive enough to be considered valid, and can take the appropriate action (application defined fault tolerance).

If the function fails, then the connection is in a bad state, and one of the following will occur:

- If *path->attributes.abort_on_failure* is *false*, then *false* is returned, and an error message is stored in *path->attributes.error_message*. The path is in a bad state and needs to be destroyed with *takyonDestroy*()
- If *path->attributes.abort_on_failure* is *true*, then an error message is printed to stderr, and then the process will call abort().

takyonDestroy

```
    char *takyonDestroy(TakyonPath **path_ret)
```

This will close the path for this endpoint and free any resources that were allocated by *takyonCreate()*. If Takyon allocated any data buffers for this path when *takyonCreate*() was called, then those buffers will be freed.

For connected interconnects, if the path is still in a good state, this will block until both endpoints coordinate a proper disconnect or a timeout occurs as defined by *path->attrs.path_destroy_timeout* value. This coordinated shutdown allows any pending data on the transport to get flushed to the destination. If the path is in a bad state due to a previous error, then the connection may be closed without any coordination with the remote end point.

Once this endpoint is destroyed, a new path with the same unique identification can be created again.

If the function fails, then one of the following will occur:

- If *path->attributes.abort_on_failure* is *false*, then an error message is returned. When this error string is no longer needed by the application, then it needs to be freed by the application.
- If *path->attributes.abort_on_failure* is *true*, then an error message is printed to stderr, and then the process will call abort().

Helpful Programming Tips

Filling in the Path Attributes

There's a lot of attributes to fill in to create a path. To simplify it down to one line of code, just use the Takyon extension function:

```
takyonAllocAttributes()
```

This function comes from the C file:

```
Takyon/extensions/takyon_attributes.c
```

Setting the Interconnects and its Optional Properties

Each path needs to have a unique ID for each type of interconnect used. For IP address based interconnects, this will be an IP address and port number or if using ephemeral port numbers then an IP address and a unique ID is needed. For most other interconnects, it will just be an ID. Here are some examples:

```
InterThreadMemcpy -ID=1
InterThreadMemcpy -ID=1

InterProcessMemcpy -ID=1
InterProcessMemcpy -ID=1

InterProcessSocket -ID=1
InterProcessSocket -ID=1

# Uses ephemeral port number (assigned by system)
Socket -client -IP=127.0.0.1 -ID=1
Socket -server -IP=127.0.0.1 -ID=1

# Uses specific port number (assigned by user)
Socket -client -IP=127.0.0.1 -port=12345
Socket -server -IP=127.0.0.1 -port=12345 -reuse
```

Since Takyon uses a text string to define the interconnect and its properties, it makes it very flexible, and can work with many different variations.

Buffer Allocations

Unless there are some specific needs for buffer memory, just let Takyon create the memory. This reduces the complexity of the application source code. This is achieved by using zero in all the elements of the send and recv addresses passed into the create() call. For example:

```
size_t sender_addr_list[3] = { 0, 0, 0 };
attrs.sender_addr_list     = sender_addr_list;
size_t recver_addr_list[2] = { 0, 0 };
attrs.recver_addr_list     = recver_addr_list;
```

If multiple Takyon paths will be sending data to a common memory buffer (e.g. a gather operation), or sending data from a common memory buffer (e.g. a scatter operation) then the application needs to allocate a memory block prior to creating the paths, and provided the appropriate memory address when setting up the address lists in the attributes structure.

Getting Helpful Print Messages

During development, make sure error reporting is turned on:

```
attributes.verbosity = TAKYON_VERBOSITY_ERRORS;
```

If errors or odd behavior are occurring, then turn on messages for each call to Takyon:

```
attributes.verbosity = TAKYON_VERBOSITY_ERRORS | TAKYON_VERBOSITY_CREATE_DESTROY |
TAKYON_VERBOSITY_SEND_RECV;
```

If tracking down difficult errors, it may be useful to turn on more details:

```
attributes.verbosity = TAKYON_VERBOSITY_ERRORS | TAKYON_VERBOSITY_CREATE_DESTROY |
TAKYON_VERBOSITY_SEND_RECV | TAKYON_VERBOSITY_CREATE_DESTROY_DETAILS | TAKYON_VERBOSITY_SEND_RECV_DETAILS;
```

When development is complete, but you still want error messages, use:

```
attributes.verbosity = TAKYON_VERBOSITY_ERRORS;
```

If your application is designed to handle communication failures in a graceful way, you may want to turn off error messages, so in that case use:

```
attributes.verbosity = TAKYON_VERBOSITY_NONE;
```

Handling Errors

If the application is not designed for handling communication errors, and you want to avoid error checking, then use the following:

```
attributes.abort_on_failure = true;
```

In this case, the application will just abort if it detects an error, and a helpful error message will be printed so you know what happened.

If you need to handle communication errors without aborting, then you need to set the following:

```
attributes.abort_on_failure = false;
```

All return values from the Takyon APIs will need to be checked.

If exit() is preferred over aborting, then the application can get the status of each Takyon call and if an error occurs call exit(<n>) with the appropriate value for <n>.

Takyon Open Source Extensions

The Takyon extension functions described in this section are provided as open source to be compiled and linked into the application as needed. These convenience APIs are provided as source code instead of libraries in order to let the developers duplicate and modify the source to best fit the application needs. It also has the added benefit of simplifying certification if needed.

An application using the Takyon extension functions needs to:

- Include "**takyon_extensions.h**", located in "Takyon/extensions/".
- Compile and link the appropriate extension C files, located in "Takyon/extensions/".

Endian (takyon_endian.c)

Helpful functions if the endian of all endpoints in the distributed application is not the same.

takyonEndianIsBig

```
bool takyonEndianIsBig()
```

Returns true if this endpoint is big endian, otherwise returns false indicating little endian.

takyonEndianSwap2Byte

```
void takyonEndianSwap2Byte(void *data, uint64_t num_elements)
```

Do byte swapping on a list of 16 bit values pointed to *data*. The number of elements in the list is *num_elements*. This will endian swap all 2 byte datatypes; e.g. short, unsigned short.

takyonEndianSwap4Byte

```
void takyonEndianSwap4Byte(void *data, uint64_t num_elements)
```

Do byte swapping on a list of 32 bit values pointed to *data*. The number of elements in the list is *num_elements*. This will endian swap all 4 byte datatypes; e.g. float, int, unsigned int.

takyonEndianSwap8Byte

```
void takyonEndianSwap8Byte(void *data, uint64_t num_elements)
```

Do byte swapping on a list of 64 bit values pointed to *data*. The number of elements in the list is *num_elements*. This will endian swap all 8 byte datatypes; e.g. double, long long, unsigned long long.

Time (takyon_time.c)

Helpful time related functions.

takyonSleep

```
void takyonSleep(double seconds)
```

Put the thread to sleep for the specified amount of time.

takyonTime

```
double takyonTime()
```

Returns the current number of seconds that have elapsed since the very first call to takyonTime().

Named Memory Allocation (takyon_mmap.c)

Functions to allocate named memory that can be accessed by remote processes in the same OS.

If inter-process memory map interconnects are used, then it may be helpful to have the application define one or more named memory blocks that can be used by the Takyon paths.

This may be especially useful if a gather type collective is used by multiple communication paths where one or more of the paths is an inter-process memory map interconnect, and the gathered data needs to be in one contiguous memory block.

takyonMmapAlloc

```
void takyonMmapAlloc(const char *map_name, uint64_t bytes, void **addr_ret, TakyonMmapHandle
*mmap_handle_ret)
```

Create a named memory block with the name *name*, and with *bytes* bytes. The address is returned in *addr_ret*, and the memory map handle is returned in *mmap_handle_ret*. If the memory map already exists, it will be freed and re-allocated.

takyonMmapFree

```
void takyonMmapFree(TakyonMmapHandle mmap_handle)
```

Frees the named memory block previously allocated by takyonMmapAlloc().

Path Attributes (takyon_attributes.c)

Simplify setting up the attributes for a Takyon path. This is not an all-encapsulating function to handle all path attribute variations, but if does handle simple symmetric paths.

takyonAllocAttributes

```
TakyonPathAttributes takyonAllocAttributes(bool is_endpointA, bool is_polling, int nbufs_AtoB, int
nbufs_BtoA, uint64_t bytes, double timeout, const char *interconnect)
```

This fills in the attributes with the specified values. Up to four lists (sender_max_bytes_list, recver_max_bytes_list, sender_addr_list, and recver_addr_list) are allocated based on the number of buffers. The address lists are set to 0 which means Takyon will allocate the buffer memory.

After this call, the attribute structure can still be modified. For example:

- Change an address value from 0 to some address value that the application already pre-allocated.
- Change one or more of the timeout values.

takyonFreeAttributes

```
void takyonFreeAttributes(TakyonPathAttributes attrs)
```

This will free any of the lists in the attribute structure that are not NULL.

Takyon Graph Description Files (takyon_graph.c)

Background

Distributed processing has two requirements:

- Some combination of multiple threads/processes/processors
- Communication between the various threads/processes/processors

Furthermore, distributed processing can significantly benefit from:

- Organized collective communications: barrier, scatter, gather, reduce, etc.
- Endian conversion functionality, for endpoint pairs with different endianness

Embedded HPC algorithm designers are typically experts in a particular domain (e.g. radar, signal processing) but are not typically experts in distributed processing. Why is this? Distributed processing is not needed to validate algorithm functionality, but instead it is generally only needed to improve performance (e.g. achieve real-time). Therefor, algorithm developers may not build in the distributed dataflow until the algorithm is functionally complete. As the distributed dataflow is designed into the algorithm (even if from the start of the algorithm design) the implementation may be tedious since multiple variations may be needed to determine best performance. Therefor it would not be recommended to hardcode a specific dataflow pattern into the application source code.

Part of Takyon's extended functionality was designed to make designing distributed dataflow easy and independently of source code. I.e. the application should not need to be recompiled in order to modify the dataflow or the properties of the communication paths. This is achieved via Takyon's graph file, a text configuration file, which describes the distributed dataflow.

Graph Based Examples

Before describing the details of graph files, it may be helpful to know what the graph based examples are so they can be referenced as you continue reading through this guide:

- **hello_world_graph**: the simplest example
- **barrier**: a tree-based barrier
- **pipeline**: processing done in a series of concurrent steps
- **reduce**: distributed 'max' value
- **scatter_gather**: paths are shared to scatter, process, then gather the results

Takyon Graph File Components

A Takyon graph file describes the following five sections:

- Processing Groups
- Processes
- Memory Buffers
- Communication Paths
- Collective Groups

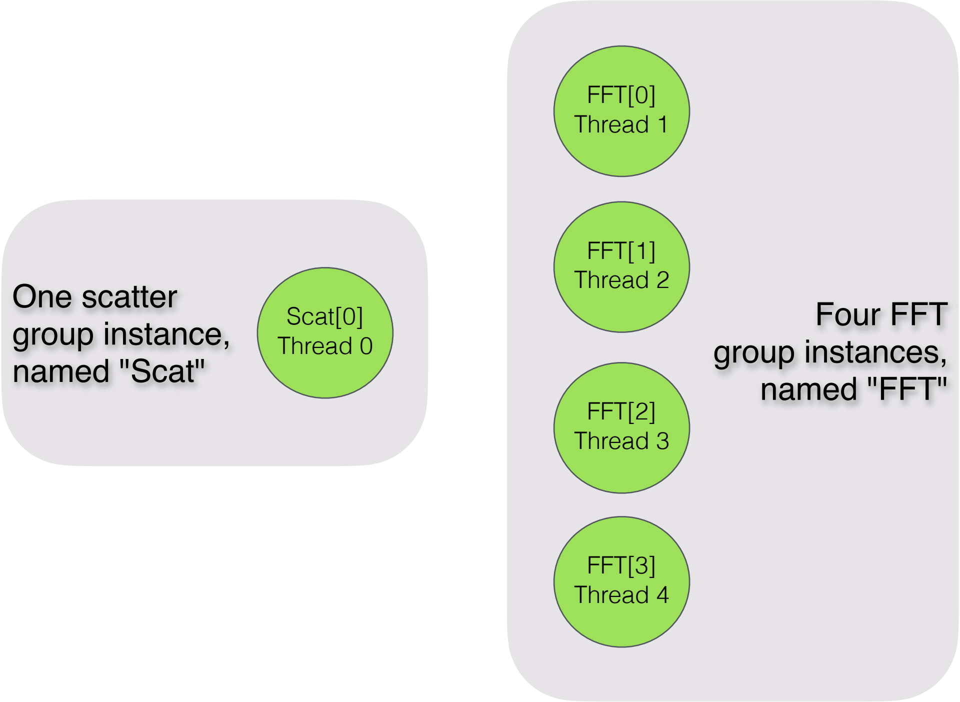
The following subsections explain the details.

Processing Groups

This section defines all the (Posix thread) endpoints in the distributed dataflow. Each group instance does two things:

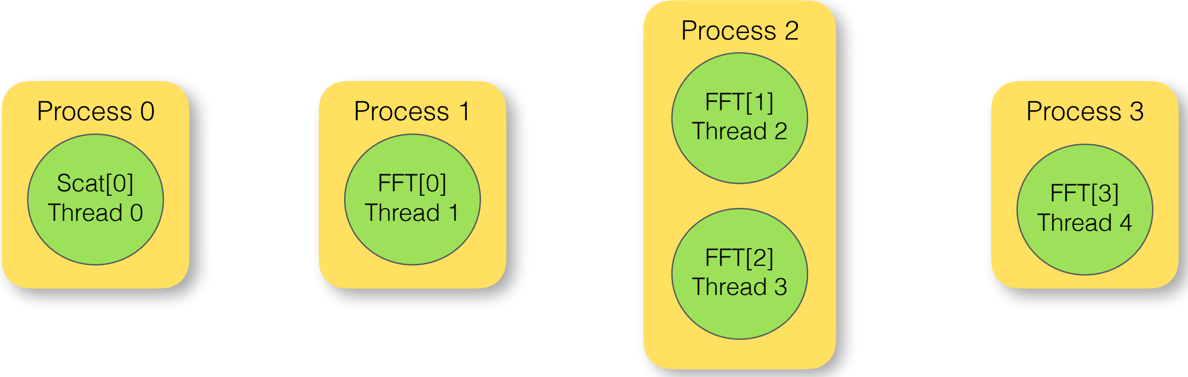
- Provide an endpoint of one or more send/receive transfers
- Do a subset of the application's overall algorithm processing; e.g. FFT, convolution,

To maintain flexibility and scalability, these endpoints should be designed to be "location independent" if possible, i.e. allowed to run on any processor/process/OS. In the diagram below, only threads are shown with no relationship to processes. The thread (group) ID is global, and each thread is assigned to a group each with a group instance index starting from 0.

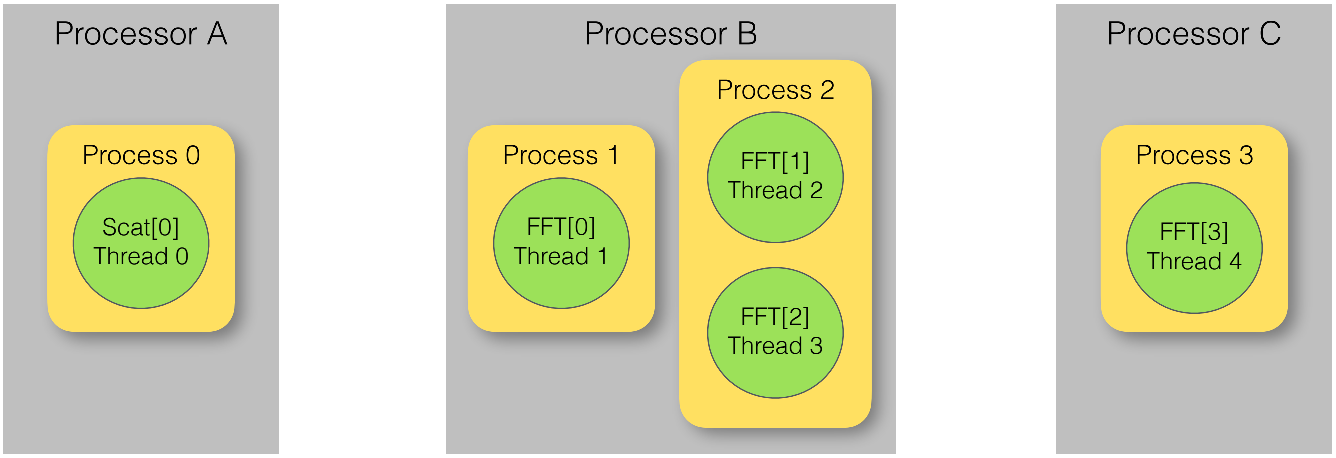

One scatter group instance, named "Scat"
Scat[0] Thread 0
FFT[0] Thread 1
FFT[1] Thread 2
FFT[2] Thread 3
FFT[3] Thread 4
Four FFT group instances, named "FFT"

Processes

This section defines all the processes in the application and maps the group instances (Posix threads) to the appropriate processes. Each process would be run as an executable and will start the appropriate group instances in an organized way such that communication paths are instantiated before the group instances are ready to do application processing. To maintain flexibility and scalability, these processes should be designed to be location independent if possible, i.e. allowed to run on any processor/OS.



The processes can be mapped to processors in various ways:

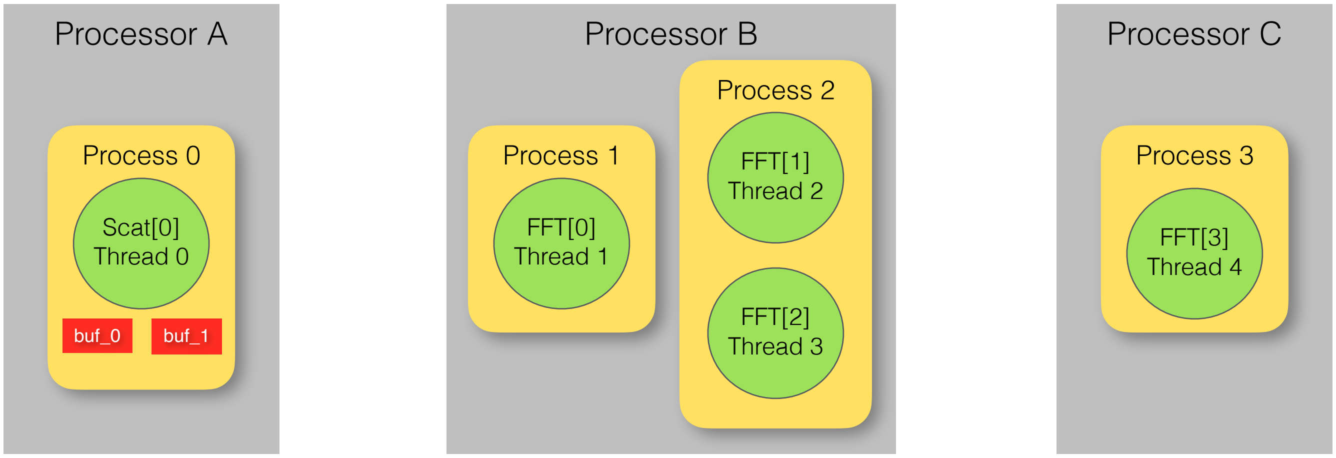


Memory Buffers

By default, Takyon implicitly allocates the transport memory buffers, but there are cases where the application needs to provide the pre-allocated transport memory buffers to Takyon before the communication path is established. This section of the graph file defines the application provided transport memory buffers that can be used by the Takyon paths. This is helpful with a few cases:

- If an endpoint is an IO device, then the IO device may be required provided the specialized memory buffers that are used by the communication path.
- Collective communications are sometimes more organized if the communication paths in the collective are sharing a single contiguous buffer. For example, a scatter's source could distribute sections of a video image to different destinations. A collective gather could collect sections of data into a single video image.

In the diagram below, the red rectangles represent application allocated memory buffers:

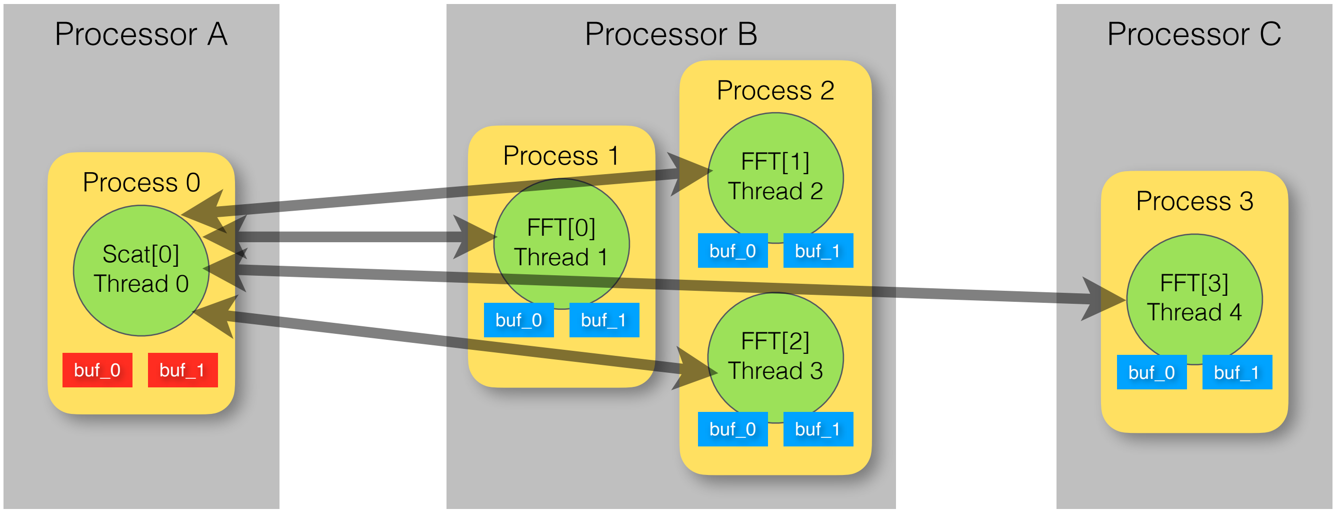


Communication Paths

This section defines all the application's communication paths and all their properties. All paths are independent of each other, so this section does not define any relationships between the paths. A path's endpoints are handled in one of the two ways:

- Both endpoints are within the Takyon application: Each endpoint of the path is associated with Takyon group instances (must be two different instances).
- Only one endpoint is within the Takyon application: Only one endpoint of the path is associated with a Takyon group instance. Some communication interconnects are one sided such as a multicast send or a multicast receive, where the other side of the Takyon path is unused. The other possibility a two-sided communication that has one end in the Takyon application and the other end is a remote 3rd party endpoint. This feature of Takyon makes it compatible to communicate with 3rd party applications even if the 3rd party application is not using Takyon.

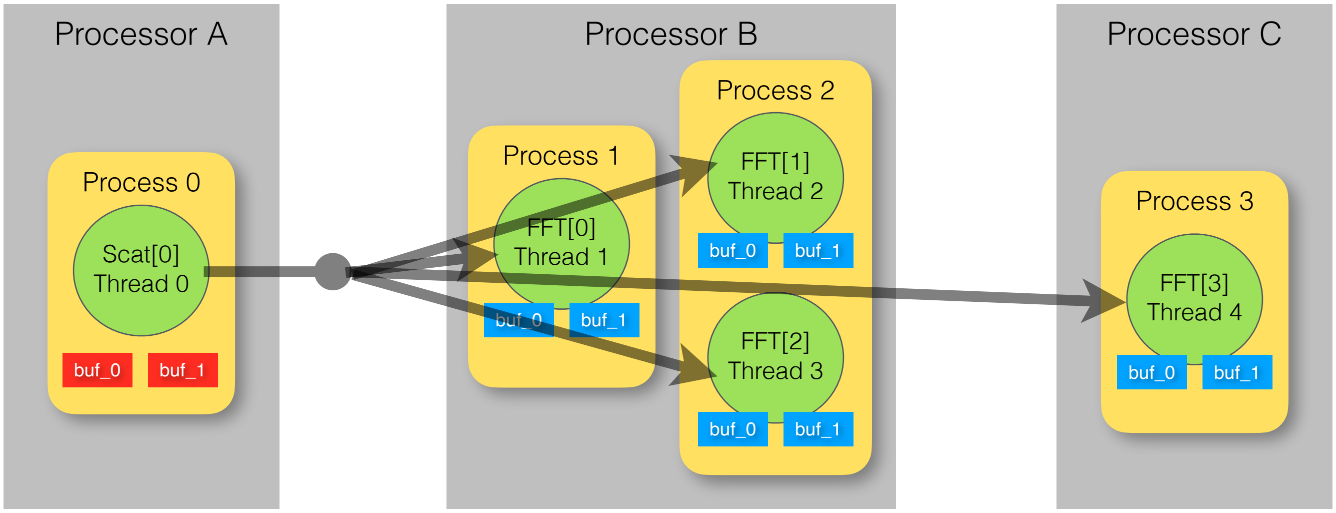
The following diagram now show the paths and includes some of the implicit transport buffers (in blue) that were automatically allocated by Takyon:



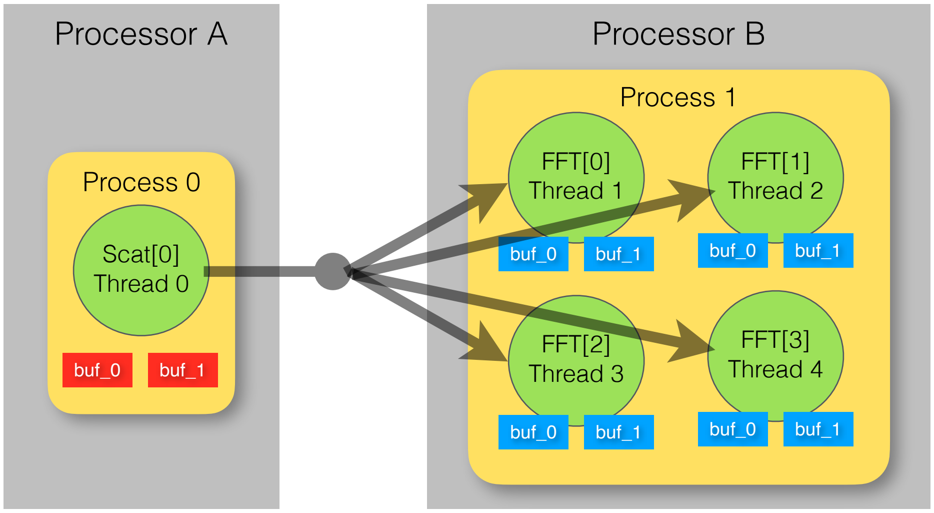
Collective Groups

Collective groups are used to create an organized relationship between a subset of Takyon paths (barrier, scatter, gather, etc.). To minimize the number of paths created, a single path can be in more than one collective group. Therefor collectives are only logical collections and do not implicitly create duplicate paths. Note that when paths are shared between collectives, the application must use the shared paths in an organized way to avoid issues with the dataflow.

In the diagram below, the four independent bi-directional paths (represented in the previous diagram) now look like an organized uni-direction one-to-many collective:



Remember that the groups and paths are independent of process/processor, which means this could easily be remapped to a different process/processor layout:



Using Graph Files

The Takyon extension functions include functionality to load, parse, and query the graph file. The design of the application processes should follow like so:

1. Implement the main() process:
   A. Load Takyon graph
   B. Create/initialize any process level non-Takyon resources needed by this process
   C. Start the Takyon group instances (Posix threads) that are mapped to this process
   D. Wait for the group instance threads to exit
   E. Free any process level non-Takyon resources needed by this process
   F. Free the Takyon graph resources
2. Implement appAllocateMemory() and appFreeMemory() needed by the main() process.
3. Implement the Takyon group Instances (the Posix threads):
   A. Create Takyon paths (one side of the path) associated with this group instance
   B. Enter and run the core processing loop; a subset of the application algorithm
   C. Destroy the Takyon paths associated with this group instance

If the above steps are implemented properly, the application should be flexible enough to run many graph file variations. I.e. run the executable on each process like so:

```
    COMMPAND_SHELL > application.exe <process_ID> <graph_file>
```

For example, if the graph file defines 5 processes, then the above command would need to be run from 5 different command shells on the appropriate processors.

A major benefit of designing this flexibility into the application source code is the application only needs to be compiled once, and the application can be distributed in many ways (via different graph files) to validate and find best performance.

The remaining sub-sections with provide the details of how to design the application source code for use with the Takyon graph files.

Step 1A main(): Load Graph File

Define the global at the top of the file:

```
static TakyonGraph *L_graph = NULL;
```

Load and parse the graph file into a data structure so it can be reference throughout the application.

```
// Load graph file and create any memory blocks
int process_id = atoi(argv[1]);     // Arg 1 from the command line
const char *filename = argv[2];     // Arg 2 from the command line
printf("Loading Takyon graph file '%s'...\n", filename);
L_graph = takyonLoadGraphDescription(process_id, filename);
takyonPrintGraph(L_graph);          // A quick way to validate loaded results
if (process_id >= L_graph->process_count) {
  printf("ERROR: No threads defined for this process id = %d\n", process_id);
  exit(EXIT_FAILURE);
}
```

The L_graph data structure is defined in Takyon/extensions/takyon_extensions.h. This is not a trivial data structure, so many convenience functions are provided to get the needed information.

When the graph is loaded, the memory buffers defined in the graph will also be allocated. This guarantees that the buffers exist before any of the Takyon group instances have started. The function to allocate the memory buffers is described below in 'Step 2'.

Step 1B main(): Allocate Custom Process-Level Resources

This is only needed if the application has some non-Takyon resources that need to be allocated before all the Takyon group instances are started. For example, a high-speed thread-safe math library that needs to be initialized before any threads use it:

```
// Init thread-safe math functionality
myMathInit();
```

Step 1C main(): Start Takyon Group Instances

Each group instance is a Posix thread. Only the group instances that are a child of the current process, identified by process_id, should be started:

```
// Start the threads
printf("Starting threads...\n");
for (int i=0; i<L_graph->process_list[process_id].thread_count; i++) {
  TakyonThread *thread_info = &L_graph->process_list[process_id].thread_list[i];
  pthread_create(&thread_info->thread_handle, NULL, thread_entry_function, thread_info);
}
```

No matter what group the instance belongs to, that same thread function, thread_entry_function(), is started. The particular type of 'Takyon group' to process is determined within the thread.

Step 1D main(): Wait For Threads to Complete

The previous step started all the threads. Now is time for the main() process to sleep until all the threads are done processing; i.e. the application is done and ready to exit.

```
// Wait for the threads to complete
for (int i=0; i<L_graph->process_list[process_id].thread_count; i++) {
  TakyonThread *thread_info = &L_graph->process_list[process_id].thread_list[i];
  pthread_join(thread_info->thread_handle, NULL);
}
```

Step 1E main(): Release Custom Resources:

If any custom resources were allocated in Step 1b, then this is the proper time to release those resources.

```
// Free custom resources
myMathFinalize();
```

Step 1f main(): Release Takyon Graph Resources

Since all the group instance threads are no longer running, it is now safe to free the graph resources.

```
// Free memory buffers and the graph data structure
takyonFreeGraphDescription(L_graph, process_id);
```

The memory buffers are deallocated using a custom defined function, described below in 'Step 2'.

At this point the process can exit with the proper code.

Step 2: Custom Memory Management

The application must define two functions even if not used:

- appAllocateMemory()
- appFreeMemory()

These two functions are used by the Takyon Graph functions in the case that memory buffers are defined in the graph file. These are only used at the process level; appAllocateMemory() is called implicitly by takyonLoadGraphDescription() and appFreeMemory() is implicitly called by takyonFreeGraphDescription().

In the graph file, each memory buffer is defined with the following parameters:

- **Name**: the name of the memory buffer using the format <name><buffer_index>. <buffer_index> must be zero or greater. This name will be referenced in the graph file in the 'Paths' section for the attribute 'SenderAddrList' or 'RecverAddrList'.
- **Process**: The process that should allocate the memory.
- **Where**: The name of what type of memory to allocate from; e.g. "CPU", "SMEM", "CUDA:0", etc. These names are not parsed by Takyon, but are application defined and handled in appAllocateMemory().
- **Bytes**: the number of bytes to allocate.

The application defines the implementation of the two memory functions.

Example of appAllocateMemory():

```
void *appAllocateMemory(const char *name, const char *where, uint64_t bytes, void **user_data_ret) {
  if (strcmp(where, "CPU") == 0) {
    // Allocate CPU memory
    *user_data_ret = NULL;
    void *addr = malloc(bytes);
    return addr;
  } else if (strncmp(where, "CUDA:", 5) == 0) {
    // Allocate CUDA memory
    *user_data_ret = NULL;
    int cuda_device_id;
    sscanf(where, "CUDA:%d", &cuda_device_id) != 1);
    cudaSetDevice(cuda_device_id);
    void *addr = NULL;
    cudaMalloc(&addr, bytes);
    return addr;
  } else if (strcmp(where, "MMAP") == 0) {
    // Allocate memory that can be shared by different processes
    char map_name[TAKYON_MAX_MMAP_NAME_CHARS];
    snprintf(map_name, TAKYON_MAX_MMAP_NAME_CHARS, "%s", name);
    TakyonMmapHandle mmap_handle;
    void *addr = NULL;
    takyonMmapAlloc(map_name, bytes, &addr, &mmap_handle);
    *user_data_ret = mmap_handle;
    return addr;
  }
  return NULL;
}
```

Associated appFreeMemory():

```
void appFreeMemory(const char *where, void *user_data, void *addr) {
  if (strcmp(where, "CPU") == 0) {
    free(addr);
  } else if (strncmp(where, "CUDA:", 5) == 0) {
    cudaFree(addr);
  } else if (strcmp(where, "MMAP") == 0) {
    takyonMmapFree((TakyonMmapHandle)user_data);
  }
}
```

Step 3 thread(): The Group Instance Thread

The thread is used to allocate the Takyon paths, run the ‘Group’ specific algorithm, and then destroy the Takyon paths. Here’s an example of a thread that can is implemented to handle three types of ‘Group’ processing:

```
static void *thread_entry_function(void *user_data) {
  // Create Takyon paths
  TakyonThread *thread_info = (TakyonThread *)user_data;
  takyonCreateGroupPaths(L_graph, thread_info->group_id);

  // Run correct threads
  TakyonGroup *group = takyonGetGroup(L_graph, thread_info->group_id);
  if (strcmp(group->name, "generate")==0) {
    generateTask(L_graph, thread_info->group_id);
  } else if (strcmp(group->name, "filterWindowingAndFFT")==0) {
    firAndFftTask(L_graph, thread_info->group_id);
  } else if (strcmp(group->name, "output")==0) {
    outputTask(L_graph, thread_info->group_id);
  } else {
    printf("Could not find correct task to run in thread\n");
    exit(EXIT_FAILURE);
  }

  // Destroy Takyon paths
  takyonDestroyGroupPaths(L_graph, thread_info->group_id);
  return NULL;
}
```

The next sub sections will describe the above source code in detail.

Step 3A thread(): Create Takyon Paths Associated with Group Instance

Use the function takyonCreateGroupPaths() to create all the Takyon path endpoints used by the specified group instance, identified by thread_info->group_id:

```
  // Create Takyon paths
  TakyonThread *thread_info = (TakyonThread *)user_data;
  takyonCreateGroupPaths(L_graph, thread_info->group_id);
```

The paths are created in an order to avoid deadlock.

Once the paths are created, the group instance is now ready to start processing.

Step 3B thread(): Run Group Instance Processing

The application will have one or more unique Takyon groups, each with their own instances. Make sure to run the proper group processing, identified by the text name group_name, with the correct instance number, identified by thread_info->group_id. For examples:

```
// Run correct threads
TakyonGroup *group = takyonGetGroup(L_graph, thread_info->group_id);
if (strcmp(group->name, "generate")==0) {
  generateTask(L_graph, thread_info->group_id);
} else if (strcmp(group->name, "filterWindowingAndFFT")==0) {
  firAndFftTask(L_graph, thread_info->group_id);
} else if (strcmp(group->name, "output")==0) {
  outputTask(L_graph, thread_info->group_id);
} else {
  printf("Could not find correct task to run in thread\n");
  exit(EXIT_FAILURE);
}
```

It’s important to note that the thread_info->group_id is not an instance index, but instead a unique number across all groups. The function takyonGroupInstance() can be used to determine, from the group ID, the index of this thread’s group:

```
int group_index = takyonGetGroupInstance(graph, group_id);
```

If the group uses collective communications, you can also use the Takyon extension functions to query the collectives and do collective transfers. See the section below ‘Collectives’ to learn more.

Step 3C thread(): Destroy the Takyon Paths

Once the group instance processing is done, the Takyon paths can be destroyed in preparation of exiting the thread.

```
// Destroy Takyon paths
takyonDestroyGroupPaths(L_graph, thread_info->group_id);
```

Graph File Format

The file format is as follows:

```
Groups
Group: <name>
  Instances: <integer >= 1>

Processes
Process: <integer_ID>
  GroupIDs: <group_name>[<index>] ...

Buffers
Buffer: <name><buffer_index>
  ProcessId: <ID>
  Where: <app_defined_name>
  Bytes: <integer >= 1>

Paths
Defaults
  IsPolling: {true | false}, {true | false}
  AbortOnFailure: {true | false}, {true | false}
  Verbosity: <or'ing of verbosity flags>, <or'ing of verbosity flags>
  PathCreateTimeout: <takyon timeout value>, <takyon timeout value>
  SendStartTimeout: <takyon timeout value>, <takyon timeout value>
  SendFinishTimeout: <takyon timeout value>, <takyon timeout value>
  RecvStartTimeout: <takyon timeout value>, <takyon timeout value>
  RecvFinishTimeout: <takyon timeout value>, <takyon timeout value>
  PathDestroyTimeout: <takyon timeout value>, <takyon timeout value>
  SendCompletionMethod: {TAKYON_BLOCKING | TAKYON_IS_SEND_FINISHED}, {TAKYON_BLOCKING |
TAKYON_IS_SEND_FINISHED}
  RecvCompletionMethod: TAKYON_BLOCKING, TAKYON_BLOCKING
  NBufsAtoB: <integer >= 0>, <integer >= 0>
  NBufsBtoA: <integer >= 0>, <integer >= 0>
  SenderMaxBytesList: <space separated list of integers >= 0>, <space separated list of integers >= 0>
  RecverMaxBytesList: <space separated list of integers >= 0>, <space separated list of integers >= 0>
  SenderAddrList: <space separated list of NULL and/or <mem_name>:<offset>>, <list of NULL and/or
<mem_name>:<offset>>
  RecverAddrList: <space separated list of NULL and/or <mem_name>:<offset>>, <list of NULL and/or
<mem_name>:<offset>>
Path: <integer ID>
  Endpoints: <group_name>[<index>], <group_name>[<index>]
  InterconnectA: <interconnect specification>
  InterconnectB: <interconnect specification>
  <other attributes to override defaults>

Collectives
Collective: <name>
  Type: { BARRIER | REDUCE | ONE2ONE | SCATTER | GATHER }
  PathSrcIds: <space separated list of <pathID>:{A|B}>
```

All section headers must be defined.

The following sections must include one or more items: Groups, Processes, Paths

The 'Defaults' section is a convenience to avoid duplicate information across multiple paths. This section does not need to be specified, but if it is, it does not need to specify all path defaults. Each item in 'Defaults' can be in any order. Each time an item is specified in 'Defaults', it replaces what the previous default was. There can be multiple 'Defaults' specified, which is helpful if groups of paths have different defaults.

One sided paths (connected and connectionless) can also be defined. When setting a path's "Endpoints:". Just define one of the sides with "-". The corresponding "interconnectA:" or "interconnectB:" will also need to be set to "-". All other parameters for the unused endpoint will be ignored.

If NBufsAtoB or NBufsBtoA is zero, then use "-" for the appropriate values for *MaxBytesList and *AddrList.

Helpful Details:

- Indentations are not required
- Comments are allowed by using a '#' as the first character on a line
- One line per definition)name, value pair. Some values are are multiple items, mostly separated by a space and to distinguish between endpoints A and B, separates by a comma.

Process level graph description functions:

takyonLoadGraphDescription

```
TakyonGraph *takyonLoadGraphDescription(int process_id, const char *filename)
```

Load graph description from a file with *filename*, and store the results in the return data structure. This should be called by all defined processes, where the *process_id* is a logical number defined in the description file. Any memory blocks defined will be allocated by the appropriate process. The application is required to implement the memory allocation functions:

```
void *appAllocateMemory(const char *name, const char *where, uint64_t bytes, void **user_data_ret)
void appFreeMemory(const char *where, void *user_data, void *addr)
```

These functions provided the flexibility for the application to allocated from any type of memory; e.g. CPU ram, MMAPs, GPU ram, IO device memory, etc. The argument *name* is used to provided a name with the memory block if needed; e.g. name memory map. The argument *where* is an application defined name, and the application can use a string comparison to determine where the memory should be allocated; e.g. "CPU", "GPU", MMAP". The argument *bytes* is the number of bytes to be allocated. The pointer to user_data_ret allows any memory handle to be passed back via a void *; e.g. named memory map handles need to be later used to free the memory.

After loading the graph description, the application should then create the threads (group instances) defined for this process, and then each thread (group instance) should call *takyonCreateGroupPaths*() to create the Takyon paths associated with the thread (group instance). Note that each group instance has a global ID and an instance ID for the named group that it's in. See the graph based examples for a clear understanding of use.

takyonFreeGraphDescription

```
void takyonFreeGraphDescription(TakyonGraph *graph, int process_id)
```

Frees any resources allocated by *takyonLoadGraphDescription*().

takyonPrintGraph

```
void takyonPrintGraph(TakyonGraph *graph)
```

This prints, to stdout, the basic information of the graph description. It may be helpful display the graph details after loading the graph.

Thread (group instance) level graph description functions:

takyonCreateGroupPaths

```
void takyonCreateGroupPaths(TakyonGraph *graph, int group_id)
```

Create all Takyon paths and collectives associated with this group instance (thread).

takyonDestroyGroupPaths

```
void takyonDestroyGraphPaths(TakyonGraph *graph, int group_id)
```

Destroy all Takyon paths associated with this group instance (thread). The collectives should be explicitly destroyed by the thread when the thread is done with the collective communications. See the graph based examples for a clear understanding of use.

takyonGetGroup

```
TakyonGroup *takyonGetGroup(TakyonGraph *graph, int group_id)
```

Given a group ID *group_id*, returns the group (thread) that it belongs to. The group can then be used to get the name of the group and the number of instances (threads) in the group.

takyonGetGroupInstance

```
int takyonGetGroupInstance(TakyonGraph *graph, int group_id)
```

Given a group ID *group_id*, return the instance index, starting with 0, in the group that it belongs to. Note that *group_id* is global across all groups.

Collective functions (to be called at the thread level after takyonCreateGroupPaths() has been called).

The following functions are not the actual collective functions that invoke communication transfers, but instead are the functions to get a handle to the collective groups defined in the graph file.

takyonGetBarrier

```
TakyonCollectiveBarrier *takyonGetBarrier(TakyonGraph *graph, const char *name, int group_id)
```

Returns the barrier collective group with the name *name* from the context of the thread with the group ID *group_id*. The parent and children Takyon paths in the returned collective will be valid paths previously created by the thread with the group ID *group_id*. This call must only be made after the thread calls *takyonCreateGroupPaths*().

takyonGetReduce

```
TakyonCollectiveReduce *takyonGetReduce(TakyonGraph *graph, const char *name, int group_id)
```

Returns the reduce collective group with the name *name* from the context of the thread with the group ID *group_id*. The parent and children Takyon paths in the returned collective will be valid paths previously created by the thread with the group ID *group_id*. This call must only be made after the thread calls takyonCreateGroupPaths().

takyonGetOne2One

```
TakyonCollectiveOne2One *takyonGetOne2One(TakyonGraph *graph, const char *name, int group_id)
```

Returns the one-to-one collective group with the name *name* from the context of the thread with the group ID *group_id*. The source and destination Takyon paths in the returned collective will be valid paths previously created by the thread with the group ID *group_id*. This call must only be made after the thread calls *takyonCreateGroupPaths*().

takyonGetScatterSrc

```
TakyonScatterSrc *takyonGetScatterSrc(TakyonGraph *graph, const char *name, int group_id)
```

Returns the source side of the scatter collective group with the name *name*. The group ID *group_id* must be the thread that is defined as the source of the scatter. This call must only be made after the thread calls *takyonCreateGroupPaths*().

takyonGetScatterDest

```
TakyonScatterDest *takyonGetScatterDest(TakyonGraph *graph, const char *name, int group_id)
```

Returns one of the destination endpoints of the scatter collective group with the name *name*. This should be called for all the destinations of the scatter. For each call, the group ID *group_id* must be the thread that is defined as the destination of the scatter. This call must only be made after the thread calls *takyonCreateGroupPaths*().

takyonGetGatherSrc

```
TakyonGatherSrc *takyonGetGatherSrc(TakyonGraph *graph, const char *name, int group_id)
```

Returns one of the source endpoints of the gather collective group with the name *name*. This should be called for all the sources of the gather. For each call, the group ID *group_id* must be the thread that is defined as the source of the gather. This call must only be made after the thread calls *takyonCreateGroupPaths*().

takyonGetGatherDest

```
TakyonGatherDest *takyonGetGatherDest(TakyonGraph *graph, const char *name, int group_id)
```

Returns the destination side of the gather collective group with the name *name*. The group ID *group_id* must be the thread that is defined as the destination of the gather. This call must only be made after the thread calls *takyonCreateGroupPaths*().

Collective Groups (takyon_collective.c)

Organize previously created Takyon paths into collective groups.

IMPORTANT: to avoid the need to initialize the collective groups; i.e. calling the init() and finalize() functions, then define the collective using a Takyon graph file, and use the graph file functions to get a handle to the collective groups (defined in the above section).

The typical types of collective calls are:

- **Barrier**: a collective used for synchronization.
- **Scatter**: one source sends data to many destinations. Data can be unique for each destination.
- **Gather**: multiple sources send to one destination.
- **One to One**: a set of paths organized in a way that does not include any scatters or gathers. Useful for single lane pipelining, parallel transfers, mesh transfers, toroid transfers, etc.
- **All to All** (not yet implemented): a collection of scatter groups and gather groups working together to move data in an organized way, such as a corner turn.
- **Reduce**: a collective that combines data from all graph nodes into a single node with a reduced answer, where the application supplies the reduction function, such as add, min, max, etc.

Barrier functions:

takyonBarrierInit

```
TakyonCollectiveBarrier *takyonBarrierInit(int nchildren, TakyonPath *parent_path, TakyonPath
**child_path_list)
```

Creates a tree based barrier where each thread in the tree can have any number of children. *nchildren* represents the number of paths accessible from the calling thread, and *child_list* contains the paths. *parent_path* is the path that communicates to the parent thread, which is NULL if this thread is the top thread in the tree. See the 'barrier' example to see how it can be constructed.

takyonBarrierRun

```
void takyonBarrierRun(TakyonCollectiveBarrier *collective, int buffer)
```

Runs the barrier algorithm. The barrier is run as a tree, and uses zero bytes messages. To have different barriers using the same paths, make sure the paths have multiple buffers, where each buffer is a different barrier. The tree algorithms is processed depth first starting from the root of the tree. Once all tree nodes have entered the barrier, then the tree is release from the barrier in the opposite depth first pattern.

takyonBarrierFinalize

```
void takyonBarrierFinalize(TakyonCollectiveBarrier *collective)
```

Frees the collective structure and the child list, but does not destroy any of the Takyon paths.

Reduce functions:

takyonReduceInit

```
TakyonCollectiveReduce *takyonReduceInit(int nchildren, TakyonPath *parent_path, TakyonPath
**child_path_list)
```

Creates a tree based barrier where each thread in the tree can have any number of children. *nchildren* represents the number of paths accessible from the calling thread, and *child_list* contains the paths. *parent_path* is the path that communicates to the parent thread, which is NULL if this thread is the top thread in the tree. See the 'reduce' example to see how it can be constructed.

takyonReduceRoot

```
void takyonReduceRoot(TakyonCollectiveReduce *collective, int buffer, uint64_t nelements, uint64_t
bytes_per_elem, void(*reduce_function)(uint64_t nelements,void *a,void *b), void *data, bool scatter_result)
```

Runs the reduce algorithm. The reduce is run as a tree, and starts from the bottom of the tree. The data is reduced and passed up the tree until it reaches the top thread. The application defines the reduce operation. This function should only be called by the root thread (the top of the tree). If *scatter_result* is true, then the reduction results are passed back down the tree so all threads in the tree have the reduction result.

takyonReduceChild

```
void takyonReduceChild(TakyonCollectiveReduce *collective, int buffer, uint64_t nelements, uint64_t
bytes_per_elem, void(*reduce_function)(uint64_t nelements,void *a,void *b), bool scatter_result)
```

Runs the reduce algorithm. The reduce is run as a tree, and starts from the bottom of the tree. The data is reduced and passed up the tree until it reaches the top thread. The application defines the reduce operation. This function should be called by all the reduction threads except the root thread (the top of the tree). If scatter_result is true, then the reduction results are passed back down the tree so all threads in the tree have the reduction result.

takyonReduceFinalize

```
void takyonReduceFinalize(TakyonCollectiveReduce *collective)
```

Frees the collective structure and child list, but does not destroy any of the Takyon paths.

One2One functions:

takyonOne2OneInit

```
    TakyonCollectiveOne2One *takyonOne2OneInit(int npaths, int num_src_paths, int num_dest_paths, TakyonPath
    **src_path_list, TakyonPath **dest_path_list)
```

Defines a set of paths that are organized in any fashion. *npaths* defines the total number of paths in the collective. *num_src_paths* defines the number of source side paths used by the calling thread, and the paths are defined in the list *src_path_list*. *num_dest_paths* defines the number of destination side paths used by the calling thread, and the paths are defined in the list *dest_path_list*.

takyonOne2OneFinalize

```
    void takyonOne2OneFinalize(TakyonCollectiveOne2One *collective)
```

Frees the collective structure and the source and destination lists, but does not destroy any of the Takyon paths.

Scatter functions:

takyonScatterSrcInit

```
TakyonScatterSrc *takyonScatterSrcInit(int npaths, TakyonPath **path_list)
```

Defines a set of *npaths* source paths, defined by *path_list*, that are associated with the source side of the scatter collective. This function duplicates the list, so the input list can be discarded after the call. All Takyon paths defined in the list must be created before making this call.

takyonScatterDestInit

```
TakyonScatterDest *takyonScatterDestInit(int npaths, int path_index, TakyonPath *path)
```

Defines one destination of the scatter. This must be called for all scatter destination. The number of paths in the scatter collective is set in *npaths.* The path for this destination is defined by *path.* The argument *path_index* defines the associated source path index in the *path_list* defined by the call to *takyonScatterSrcInit*(). The Takyon path defined by *path* must be created before making this call.

takyonScatterSend

```
void takyonScatterSend(TakyonScatterSrc *collective, int buffer, uint64_t *nbytes_list, uint64_t
*soffset_list, uint64_t *doffset_list)
```

Do a scatter send on the collective *collective* defined by *takyonScatterSrcInit()*. The data from each path's buffer index *buffer* will be sent. Each path has an independent number of bytes, source offset, and destination offset. The size of the three lists is *collective->npaths*, which also defines the number of scatter destinations.

takyonScatterRecv

```
void takyonScatterRecv(TakyonScatterDest *collective, int buffer, uint64_t *nbytes_ret, uint64_t
*offset_ret)
```

Do a scatter receive on the collective *collective* defined by *takyonScatterDestInit*(). This function must be called by each destination in the scatter group. The data will arrive in the path's buffer index *buffer* with the number of bytes as pointed to by **nbytes_ret*, and with an offset as pointed to by **offset_ret*.

takyonScatterSrcFinalize

```
void takyonScatterSrcFinalize(TakyonScatterSrc *collective)
```

Frees the collective structure and the path list, but does not destroy any of the Takyon paths.

takyonScatterDestFinalize

```
void takyonScatterDestFinalize(TakyonScatterDest *collective)
```

Frees the collective structure, but does not destroy the Takyon path.

Gather functions:

takyonGatherSrcInit

```
TakyonGatherSrc *takyonGatherSrcInit(int npaths, int path_index, TakyonPath *path)
```

Defines one source of the gather. This must be called for all gather sources. The number of paths in the gather collective is set in *npaths.* The path for this destination is defined by *path.* The argument *path_index* defines the associated destination path index in the *path_list* defined by the call to *takyonGatherDestInit*(). The Takyon path defined by *path* must be created before making this call.

takyonGatherDestInit

```
TakyonGatherDest *takyonGatherDestInit(int npaths, TakyonPath **path_list)
```

Defines a set of *npaths* destination paths, defined by *path_list*, that are associated with the destination side of the gather collective. This function duplicates the list, so the input list can be discarded after the call. All Takyon paths defined in the list must be created before making this call.

takyonGatherSend

```
void takyonGatherSend(TakyonGatherSrc *collective, int buffer, uint64_t nbytes, uint64_t soffset, uint64_t
doffset)
```

Do a gather send on the collective *collective* defined by *takyonGatherSrcInit*(). This function must be called by each sources in the gather group. The data from the path's buffer index *buffer* will be sent, where the number of bytes is defined by *nbytes*, the source offset is *soffset*, and the destination offset is *doffset*.

takyonGatherRecv

```
void takyonGatherRecv(TakyonGatherDest *collective, int buffer, uint64_t *nbytes_list_ret, uint64_t
*offset_list_ret)
```

Do a gather receive on the collective *collective* defined by *takyonGatherDestInit()*. The data will arrive in each path's buffer indexed by *buffer*. Each path has an independent number of received bytes and offset, as defined by the two lists *nbytes_list_ret* and *offset_list_ret*. The size of the two lists is *collective->npaths*, which also defines the number of gather sources.

takyonGatherSrcFinalize

```
void takyonGatherSrcFinalize(TakyonGatherSrc *collective)
```

Frees the collective structure, but does not destroy the Takyon path.

takyonGatherDestFinalize

```
void takyonGatherDestFinalize(TakyonGatherDest *collective)
```

Frees the collective structure and the path list, but does not destroy any of the Takyon paths.

Collectives: How to Use

If a group instance thread uses collective communications, you can also use the Takyon extension functions to query the collectives and do collective transfers.

Takyon extension include functionality for the following collectives:

- Barrier: uses a tree layout to run the barrier on Log(N) steps
- Scatter: one to many where the amount to each destination is application defined
- Gather: many to one where the amount gather from each source is application defined
- Reduce: a distributed reduction operation like min(), max(), etc.
- One-to-One: a custom layout of paths as defined by the graph file

The following sub sections discuss how to implement the collectives.

Barrier

A simple implementation of a barrier collective:

```
void barrierTask(TakyonGraph *graph, int group_id, int ncycles) {
  // Get information about the barrier
  TakyonCollectiveBarrier *barrier = takyonGetBarrier(graph, "barrier", group_id);
  int barrier_buffer = 0;
  int barrier_nbufs;
  if (barrier->parent_path != NULL) {
    // A leaf node, in the search tree, with no children
    barrier_nbufs = barrier->parent_path->attrs.nbufs_AtoB;
  } else {
    // A parent node, in the search tree, with children
    barrier_nbufs = barrier->child_path_list[0]->attrs.nbufs_AtoB;
  }

  // Run the barrier multiple times
  for (int i=0; i<ncycles; i++) {
    // Run the collective barrier
    takyonBarrierRun(barrier, barrier_buffer);
    // Go to next buffer
    barrier_buffer = (barrier_buffer + 1) % barrier_nbufs;
  }

  // Deallocate barrier resources
  takyonBarrierFinalize(barrier);
}
```

Scatter Send

A collective scatter sender defines how much data each child node receives.

```
void scatterSendTask(TakyonGraph *graph, int group_id) {
  // Get a handle to the scatter send collective
  TakyonScatterSrc *scatter_src = takyonGetScatterSrc(graph, "scatter", group_id);
  int buffer = 0;
  int nbufs = scatter_src->path_list[0]->attrs.nbufs_AtoB;
  int num_children = scatter_src->npaths;
  uint64_t parent_bytes = scatter_src->path_list[0]->attrs.sender_max_bytes_list[0];
  uint64_t child_bytes = parent_bytes / num_children;

  // Allocate some arrays for organizing the contiguous scatter buffer
  uint64_t *nbytes_list = (uint64_t *)malloc(num_children*sizeof(uint64_t));
  uint64_t *soffset_list = (uint64_t *)malloc(num_children*sizeof(uint64_t));
  uint64_t *doffset_list = (uint64_t *)malloc(num_children*sizeof(uint64_t));

  // Fill the contiguous scatter buffer
  uint8_t *send_addr = (uint8_t *)scatter_src->path_list[0]->attrs.sender_addr_list[buffer];
  for (uint64_t j=0; j<parent_bytes; j++) {
    send_addr[j] = (uint8_t)(j);
  }

  // Organize how the data is sent to the scatter destinations
  for (int i=0; i<num_children; i++) {
    nbytes_list[i] = child_bytes;
    soffset_list[i] = i*child_bytes;
    doffset_list[i] = 0;
  }

  // Send the data to the scatter destinations
  takyonScatterSend(scatter_src, buffer, nbytes_list, soffset_list, doffset_list);

  // Free the scatter resources
  takyonScatterSrcFinalize(scatter_src);
  free(nbytes_list);
  free(soffset_list);
  free(doffset_list);
}
```

Scatter Receive

Each destination of collective scatter receives data from the single sender.

```
void scatterRecvTask(TakyonGraph *graph, int group_id) {
  // Get a handle to the scatter recv collective
  TakyonScatterDest *scatter_dest = takyonGetScatterDest(graph, "scatter", group_id);
  int buffer = 0;
  int nbufs = scatter_dest->path->attrs.nbufs_AtoB;
  uint64_t child_bytes = scatter_dest->path->attrs.recver_max_bytes_list[0];

  // Wait for the segment of scatter data to arrive
  uint64_t nbytes;
  uint64_t offset;
  takyonScatterRecv(scatter_dest, buffer, &nbytes, &offset);

  // Do something with the data
  uint8_t *recv_addr = (uint8_t *)scatter_dest->path->attrs.recver_addr_list[buffer];

  // Free the scatter resources
  takyonScatterDestFinalize(scatter_dest);
}
```

Gather Send

There are multiple sources to a gather. Each instance has its own instance index.

```
void gatherSendTask(TakyonGraph *graph, int group_id) {
  // Get a handle to the gather send collective
  TakyonGatherSrc *gather_src = takyonGetGatherSrc(graph, "gather", group_id);
  int task_instance = takyonGetGroupInstance(graph, group_id);
  int buffer = 0;
  int nbufs = gather_src->path->attrs.nbufs_AtoB;
  uint64_t child_bytes = gather_src->path_list[0]->attrs.sender_max_bytes_list[0];

  // Fill in the chunk of gather data
  uint8_t *send_addr = (uint8_t *)gather_src->path->attrs.sender_addr_list[buffer];
  for (uint64_t j=0; j<child_bytes; j++) {
    send_addr[j] = (uint8_t)j;
  }

  // Send to the single gather destination.
  // This segment of data will fit into the bigger contiguous block in the gather destination buffer
  uint64_t soffset = 0;
  uint64_t doffset = task_instance * child_bytes;
  takyonGatherSend(gather_src, buffer, child_bytes, soffset, doffset);

  // Free the gather resources
  takyonGatherSrcFinalize(gather_src);
}
```

Gather Receive

Gather from multiple source, optionally into a single contiguous buffer.

```
    void gatherRecvTask(TakyonGraph *graph, int group_id) {
      // Get a handle to the gather recv collective
      TakyonGatherDest *gather_dest = takyonGetGatherDest(graph, "gather", group_id);
      int buffer = 0;
      int nbufs = gather_dest->path_list[0]->attrs.nbufs_AtoB;
      int num_children = gather_dest->npaths;

      // Allocate some arrays for organizing the contiguous gather buffer
      uint64_t *nbytes_list = (uint64_t *)malloc(num_children*sizeof(uint64_t));
      uint64_t *doffset_list = (uint64_t *)malloc(num_children*sizeof(uint64_t));

      // Wait for the gather data to arrive
      takyonGatherRecv(gather_dest, buffer, nbytes_list, doffset_list);

      // Do something with the data.
      // The receive buffers may have been organized from a single contiguous buffer
      uint8_t *recv_addr = (uint8_t *)gather_dest->path_list[0]->attrs.recver_addr_list[buffer];

      // Free the gather resources
      takyonGatherDestFinalize(gather_dest);
      free(nbytes_list);
      free(doffset_list);
    }
```

Reduce

This is used to do distributed vector reductions, such as min, max, accumulate, etc.

See the Takyon/examples/reduce/reduce.c for a working example.

One-to-One

This allows paths to be organized in arbitrary ways; e.g. mesh, pipeline.

See the Takyon/examples/pipeline/pipeline.c for an example of how to construct a pipeline type of dataflow.

Interconnect Porting Kit

Adding a new interconnect (connected, connectionless, IO device, etc) is relatively easy.

1. Copy the file `Takyon/API/src/takyon_template.c` to `Takyon/API/src/takyon_<interconnect>.c`
2. Fill in `takyon_<interconnect>.c` with the proper implementation. Use the existing source files as a reference on how to properly implement the source code.
3. If any utilities need to be added to keep the implementation broken up into manageable chunks, just add a utility file `Takyon/API/src/utils_<functionality>.c`, and add the global functions to either `Takyon/API/inc/takyon_private.h` or to `Takyon/API/inc/utils_<functionality>.h`
4. Add the interconnect and any new utility files to the appropriate makefiles in `Takyon/API/builds/*/*`.
5. Run the examples to validate the new implementation.

WE INNOVATE. WE DELIVER. YOU SUCCEED.

Americas: 866-OK-ABACO or +1-866-652-2226 **Asia & Oceania:** +81-3-5544-3973 **Europe, Africa, & Middle East:** +44 (0) 1327-359444
Locate an Abaco Systems Sales Representative visit: abaco.com/products/sales

abaco.com @AbacoSys