WHITE PAPER

AXIS Takyon: a much-needed solution to communication in embedded HPC applications

Michael Both,
Principal Software Engineer

abaco
SYSTEMS

AXIS Takyon: a much-needed solution to communication in embedded HPC applications

Introduction

Abaco is leading an effort to create a new open standard point-to-point communication in order to address a significant problem in the embedded HPC (eHPC for short) market.

The problem

In the eHPC (embedded high performance computing) market, developers are focused on domain-specific application development (e.g. radar processing, signal intelligence, autonomous driving). These domain-specific problems require substantial algorithm expertise (math, physics, etc.) not related to communication.

The resulting applications may be compute-intensive such that in order to achieve real-time, the algorithm must be distributed across multiple compute elements (e.g. CPUs, GPUs, FPGAs, ASICs) that make up a heterogeneous system. They must also integrate high throughput I/O devices like video, lidar, and radar sensors as well as high speed storage. Distributing the processing and integrating I/O currently requires multiple non-trivial communication interfaces (e.g. sockets, verbs, shared memory, semaphores, conditional variables).

Overall, eHPC developers are immersed in the algorithm development (the primary focus), but when it comes time to distributing the application, point-to-point communication is typically a secondary focus.

Why is Point-to-point Communication a Problem?

The world of computing in the eHPC market is evolving quickly. In the last 20 years, there have been great advances in hardware and software.

Some examples of how hardware has evolved:

- Multi-core processors are now commonplace, which allows for concurrent MIMD computing in a single CPU.
- General purpose processing on GPUs to allow for concurrent SIMD computing. This is excellent for image processing, deep neural network processing, etc.
- High speed interconnects like RDMA which avoid operating system calls during the transfer.
- High throughput I/O sensors and devices: e.g. HD cameras, lidars, radars, VR headsets, solid state hard drives.

AXIS Takyon: a much-needed solution to communication in embedded HPC applications

Software is also evolving in remarkable ways, making software development easier to code and easier to debug:

- Artificial Intelligence
- Compiler technologies: e.g. optimizations, OpenMP, code analysis, debuggers
- GPU languages: e.g. CUDA, OpenCL, Vulkan, Metal
- Wireless protocols: e.g. Wi-Fi, Bluetooth, NFC
- Multiple virtualized operating systems running on a single CPU
- RDMA communication (e.g. verbs, network direct)

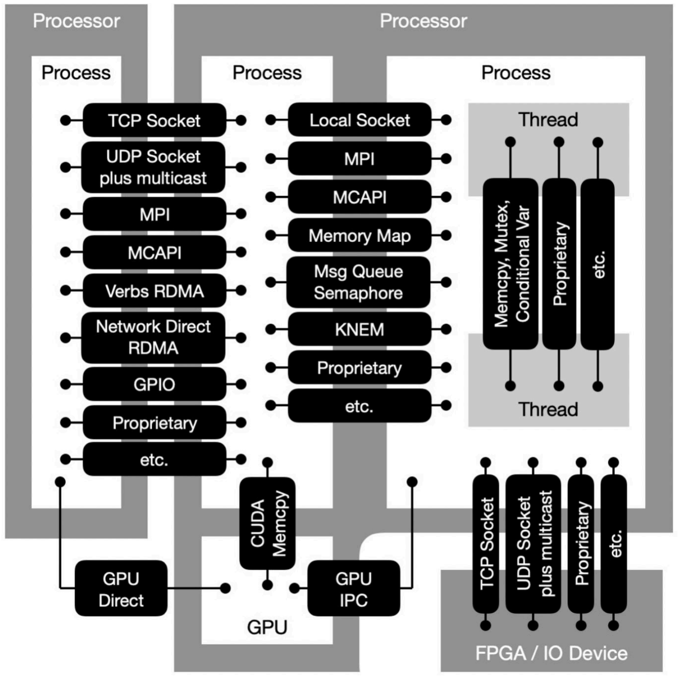
Unfortunately, when it comes to point-to-point communication APIs, there has been little advancement towards making it easier for developers who are not focused on communication - but need it. There are three fundamental issues:

- Fragmentation
- Complexity or lack of functionality
- Geriatrics

Each of these poses a significant barrier to an eHPC developer since communication is a secondary focus, not a primary focus like the core algorithm technologies with which they are familiar. This lack of wisdom for integrating communication may result in an overall algorithm performance hit and may increase development and maintenance costs. The following is a discussion of each of these issues.

Fragmentation
An eHPC application may have multiple input devices (FPGAs, cameras, radar, etc.) and multiple heterogeneous compute devices (CPUs, GPUs). To properly distribute an eHPC application, multiple point-to-point communication frameworks (open standards and proprietary) may be required.

The diagram below demonstrates the potential complexity (although in practice, it would be a subset of this).



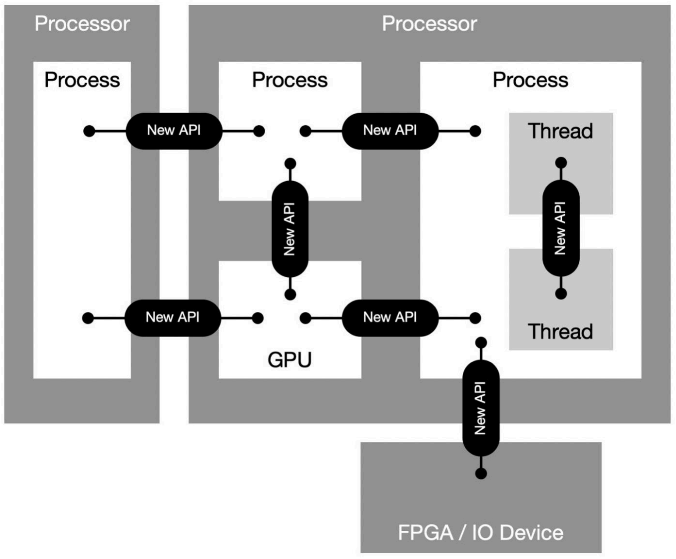
This requires multiple communication API implementations in the application source code. The more source code, the more potential bugs, the more lines of code, and the more changes needed if the application is mapped to different hardware or interconnects.

In the eHPC market, certification is common. If the application has significantly more code due to the communication, this results in increased time and cost for the certification process.

AXIS Takyon: a much-needed solution to communication in embedded HPC applications

Ideal Solution

The ideal solution would be to have all communication handled by a single API, as shown below:



Complexity or lack of functionality

Even if fragmentation was not an issue, the common point-to-point communication APIs used today are not trivial. Low level APIs (e.g. sockets, verbs) seem to follow the detailed concepts of the underlying hardware instead of being well-defined black boxes following the simple intuition of sending and receiving. High level APIs (e.g. MPI) hide the hardware details in a black box, but don't have the functionality needed by eHPC applications.

To quantify complexity, all that needs to be shown is how many realistic lines of code are needed to handle a simple transfer of data. To be realistic in an eHPC application, the count must include:

- All error checking: An eHPC application may have a critical requirement of detecting failures.
- Portable source code: An eHPC application may need to work on various operating systems. Even an API as portable as sockets, there are still a wide set of variations between Windows®, Linux®, VxWorks®, Mac, etc. that require either #ifdefs, if statements, or separate compiled source files.
- Performance-enabling source code: Applications have different requirements in terms of performance (latency, throughput, determinism). Communication paths need to be tuned for the desired performance behavior. For example, by default, sockets are set to fill a buffer before sending (not good for low latency).

With the above common eHPC restrictions, the lines of code for setting up and executing a simple one-way transfer are not trivial.

The following chart shows an intelligent estimate for common communication APIs:

P2P Communication API	One Way Communication LOC
Socket	~200
Verbs, Network Direct	~2000
Mutexes and conditions to track local memory or pointer transfers	~200
Shared memory (between processes)	~200
MPI (Not widely adopted by the eHPC community)	~20

This is just the minimum. The number of lines of code goes up as concepts like multi-buffering, CUDA interoperability, collective communication, etc. are added

For an eHPC engineer trying to distribute an algorithm across compute elements, their typical understanding of point-to-point communication is about two concepts: sending and receiving - but the engineer will find it daunting to realize common communication APIs are not just two functions: send() and receive().

AXIS Takyon: a much-needed solution to communication in embedded HPC applications

The following table shows the number of functions that should be learned to fully understand how to use the API:

P2P Communication API	API Function Count
Socket	~20
Verbs, Network Direct	~100
Mutexes & Conditions to track local memory or pointer transfers	~20
Shared memory (between processes)	~20
MPI (Not widely adopted by the eHPC community)	~300

Sockets may only have about 20 functions, but they have hundreds of attributes that can be set - so that needs to be taken into account also.

The learning curve for most communications APIs is very high. Without studying all the functionality and terminology, the engineer would not have the knowledge or wisdom to understand what functions will best satisfy the strict requirements of an eHPC application, leading to compromises in the design. Even if the engineer does find the time to really learn a complex communication API - since it's likely a secondary API - the complex details may be quickly forgotten, especially if there is a regular turnover of engineers on the project.

Ideal Solution

The ideal solution would be to have a minimal set of APIs but still provide the flexibility to the experts:

P2P Communication API	One Way Communication LOC	API Function Count
New API	~20	~5

This includes minimizing any attributes of the API to only the options that are intuitive. Any other details should be hidden or expressed in some interconnect-specific set of flags defined outside of the formal specification. For example, sockets need an IP address and port number, so a string could be used to set that up: "socket -ip 192.168.1.234 -port 1234"

Geriatrics

Communication interconnects (i.e. the hardware) have a set of capabilities that may or may not be exposed by the communication API (i.e. the software). If the software does not expose certain capabilities of the hardware, then eHPC developers may find it difficult to achieve application requirements and compromises might be made.

The following subsections describe various issues that communication APIs might have.

Fault Tolerance

Most lower level communication APIs support the hooks for fault tolerance:

- Know if a communication path has become invalid (such as disconnect detection)
- Timeouts for each stage of communication (connect, send, receive, disconnect) to know if the communication path is no longer responsive in a reasonable time
- Avoid static dataflow; i.e. should be able to create and destroy paths at any time during the life of the application
- Communication paths should be independent; i.e. if one path goes bad, it should not affect other paths

Notice that these are just hooks for the application to build in fault tolerance, as the communication API should not try to make decisions for the application. Only the application knows what 'plan B' is in the case of a failure.

Some communication APIs, such as MPI, lack explicit hooks for fault tolerance. MPI itself may be internally fault-tolerant, but this may not be helpful to an eHPC application that needs to explicitly know about failures so it can invoke that 'plan B'. MPI is also globally initialized, which means if one communication path goes down, then other paths may be brought down with it. All these issues are likely one of the primary reasons MPI is not generally used in the eHPC field.

Proposed Solution: Make sure all stages of communication (connect, send, receive, disconnect) support timeouts and failure detection and return that information back to the application. Also, make sure all communication paths are independent of each other.

AXIS Takyon: a much-needed solution to communication in embedded HPC applications

Preparing Transfer Memory Buffers

Some interconnects, like RDMA, require that transfer memory is pre-registered before use. This usually means pinning the memory buffers so they can't be swapped out to disk. Communication APIs like sockets and MPI both pass memory addresses to the send and receive functions at the time of transfer.

Following is some pseudo code to represent that concept:

```
Send(path_id, sender_data_addr, bytes);
Recv(path_id, recver_data_addr, &bytes_received);
```

This means there needs to be an operating system context switch to pin the memory on both sides of the transfer(see the Linux man page for the 'mlock()' family of functions) , then the sender needs to ask the receiver where the data will be sent, which requires an implicit round trip. Pinning memory and doing a hidden round trip before the real transfer starts will have a significant impact on latency and determinism. Pinned memory caching can be used to help reduce the problem, but it does not eliminate the problem. This is another primary reason why MPI is not typically used in the eHPC market.

Proposed Solution: Register transfer buffers outside of critical processing, like when a communication path is being created.

The above pseudo code would change to:

```
Send(path_id, bytes);
Recv(path_id, &bytes_received);
```

Where the sender and receiver data addresses would be known at the time the path was created.

Sender Synchronization

Some interconnects, like sockets, have built in synchronization to know when it is safe to send data to the receiver, i.e. the receiver has finished processing the previous block of data and is no longer using the memory buffer - so new data can safely arrive without corrupting the processing of the previous data. Some interconnects don't have this built in synchronization, so it's up to the application to know when it is safe to send data. Interconnects like MPI add implicit synchronization, but that has some side effects:

- Transfers are no longer deterministic
- Extra synchronizations may occur that were not needed, which decreases performance
- Synchronization may occur at a time that is not ideal

Proposed Solution: Only the application can know when it's the appropriate time to inform the sender that the receiver is ready for more data. This form of synchronization should therefor be left to the application via explicit means (unless built into the interconnect). For example, when the receiver has finished with a transfer buffer, then it could send a zero-byte message to the sender as a notification that the buffer is free to be filled again.

Multiple Ways to Send/Receive

Some interconnects have multiple ways of sending and receiving. For example, verbs has three fundamental ways of transferring (one-sided push, one-sided pull, two-way coordinated transfer). MPI has four variations of send/receive plus one-sided push and one-sided pull transfers. To complicate these transfer methods, each one may also have different ways to handle transfer completion, either by secondary functions, or though some user defined way such as spinning on a variable change. These non-trivial variations may be acceptable for an expert - but not for an engineer who sees communication as secondary.

Proposed Solution: Support one type of send/receive functionality. The best possible foundation for sending and receiving is to have a two-sided, one-way, zero-copy transfer where the completion notification for receiving is built into the transfer. Any other functional model of send/receive would likely add unwanted overhead and source code complexity.

Data Privacy

Some communication APIs want to know the data structure of the messages being passed. For example, MPI requires knowledge of the data structure in the sender and receiver, via MPI-defined data structures (not language-native data structures). Not only does this increase source code to convert between native data structures and MPI data structures, but this may also be an issue with privacy as messages could be reverse-engineered in the MPI transport. This can be avoided by just passing contiguous byte arrays - but then defeats the purpose of the clever features MPI allows while the data is on the transport.

But: clever data manipulation on the transport may also have performance impacts. For example, if complex data (interleaved real and imaginary values) is sent, and it's reorganized on the transport to be split into an array of real values and another array of imaginary values, then the transport must do some work to split the interleaved data. Most transports don't have this capability in hardware which means the CPU must implicitly do one of the following:

AXIS Takyon: a much-needed solution to communication in embedded HPC applications

- Allocate temporary memory on the source side to create two contiguous buffers for the split data, spend time filling the buffers, do the transfer, and finally de-allocate the temporary buffers. This may fail if the needed memory is not available.
- Send each individual value with a separate transfer. It might be thought that many interconnects support strided transfers, but in reality, almost no interconnects support this. Even InfiniBand® does not, as it only supports a small set of linked transfers (typically hundreds), which is not useful when sending thousands or millions of strided values.

In both cases, there will be a significant impact on latency, throughput, and determinism.

Proposed Solution: Don't expose the message datatype to the communication API, and don't allow non-contiguous data transfers. Applications can explicitly do the most optimal processing to handle strided data, and there will be no mystery to the application developer, who will be best at optimizing this type of situation.

Mixing Polling and Event Driven
Most communication APIs allow the option to handle send completion and receive completion via either polling or event driven. Polling is good for very low latency – but at the expense of consuming valuable CPU cycles. Event-driven is excellent for reducing unneeded CPU spinning at the expense of latency.

With most lower level communication APIs, each path can be set to polling or event-driven independent of other paths. Some high-level APIs, such as MPI, only allow one choice for all paths. This can be very restricting in an eHPC application when some data transfers need to be very responsive via polling, and other data transfers can be non-critical, allowing the CPU to undertake more critical processing if the path is event-driven. This is yet another reason why MPI may not fit well in an eHPC application.

Proposed Solution: Ensure all paths are independent of each other and allow each path to be polling or event-driven.

Why has this problem not yet been solved?
Outside of the eHPC market, point-to-point communication is generally not used in a heterogeneous environment.

In the supercomputer HPC market, MPI is prevalent. This market is more focused on, for example, simulation or solving very large math problems over a period of non-critical time. Real-time, latency, and explicit fault tolerance are generally not a concern. This means only one communication API - in this case, MPI - will suffice. It should be noted that MPI tried to create the MPI/RT initiative to better fit in the eHPC market, but this was not adopted by the eHPC community. This is likely due to the foundation of MPI not fitting with the requirements of an eHPC application.

In the mobile (iOS, Android), IoT (surveillance cameras, thermostats, etc.) or web markets, generally only one interconnect type is needed. Communication with the internet is dominated by sockets.

Markets outside of eHPC don't suffer from the fragmentation of heterogeneous applications. These other markets may have a little issue with complexity or geriatrics, but it's insignificant enough that it doesn't warrant a new standard.

This leaves only the smaller eHPC market that could significantly benefit from a new unified point-to-point communication standard. Within this market, engineers see communication as a secondary API, so they are not likely to have the wisdom or time to define a communication API that will not only work for their application, but for all other applications in the eHPC field. This makes it very difficult to find dedicated participation in the various eHPC markets to define a new standard.

AXIS Takyon: a much-needed solution to communication in embedded HPC applications

Introducing Takyon

With many years of experience using and implementing point-to-point communication APIs, Abaco has created an open source point-to-point communication specification and reference implementation, which intends to solve the issues discussed in this paper:

- One API with five functions that can support all modern interconnects including I/O device communication
- Designed for real-time, determinism, and fault tolerance
- Easy for the beginner and flexible for the expert

Takyon can be found on GitHub at:
https://github.com/Abaco-Systems/axis-takyon

This implementation includes many examples to show how simple, powerful, and flexible Takyon is. Abaco will soon create a commercial version of Takyon that will include more interconnects for easier use with GPUs, I/O devices, and RDMA-enabled hardware.

Abaco believes it has created a specification that is well suited as an open standard. This led Abaco to approach Khronos (a standards group) since they are focused on embedded/heterogeneous computing and have experience in replacing a geriatric API (OpenGL™) with something that better fits modern hardware (Vulkan).

Khronos created an exploratory group for Takyon to gauge industry interest. The exploratory group surveyed the eHPC market to find that there is a large interest in an API that solves the issues described in this paper. In order to proceed with a Khronos working group to create an open standard specification, new members are needed to become part of the group who are willing to actively participate in creating the new standard.

If you would like to participate in formulating the standard, please contact David Tetley, Khronos Exploratory Group Chair and Principal Software Engineer at Abaco Systems.

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