**Agenda**

|  |  |  |  |
| --- | --- | --- | --- |
| Tuesday Afternoon - 8/19 | | | |
| **Start** | **Duration** | **Topic** | **Notes** |
| 1:00 | :90 | Verbs extensions | Rich Graham / Liran Liss |
| 2:30 | :15 | Break |  |
| 2:45 | :90 | Pseudo-code small group session | Break into small groups |
| 4:15 | :90 | Review pseudo-coding results | Full group |
| 5:45 |  | Adjourn |  |
|  |  |  |  |

**Verbs Extensions – Rich Graham**

See the slide deck on the OFA website for more detail

Key application Concerns

* Investment preservation in verbs and verbs infrastructure
* Improvements to critical data paths
* Scalability
* Enhanced memory management
* Improved hardware capability/functionality

Optimized Data Path – proposed approach is:

* Capability query
* Direct calls to vendor – get the compiler to completely avoid fn calls, going straight to h/w
* Addressed the following: RDMA operations, Channel operations, Completion operations
* Ability to specify vairous completion options including completion counters…
* Minimal completion information whenever possible, detailed queries provided when needed.

Q: what is MLNX proposing?  
A: can’t ignore existing user base. Need a way to introduce new capabilities over time. Not clear if this is in parallel to, instead of, or an augmentation of libfabrics. Has many of the same characteristics being baked into libfabrics.

Q: Has there been any investigation into mapping this to libfabrics? i.e. extended verbs as a provider for libfabrics.

A: not clear yet.

Scalability

* Constant memory footprint enabled by some form of reliable datagram
* Asynchronous address resolution; could include an Address Vector-style abstraction

Enhanced Memory Support

* Avoid local registration via implicit On-Demand-Paging
* Register arbitrary address ranges via lazy memory registration
* User-mode memory registration
  + Define a memory key that refers to logically contiguous but physically non-contiguous memory regions
* Application-controlled R\_Key

Other functionality being considered

* Manual progress supported by polling CQs associated with QPs that require manual progress
* Relaxed ordering (?)
* Optimized completions via counter operations
* query\_interface extended verb to support vendor extensions

**Pseudo-code small groups**

MPI Point-to-Point Operations group – see the file MPI-Pt2Pt.txt

* MPI\_Send – completion ordering; conclusion is that libfabrics appears to have the right approach.
* Cancel – distinguish between Send and Receive cancel. Cancelling a Send seems rather pointless; Receive Cancel is required by MPI.
  + **AR: Is it necessary to support a query of the available cancel properties**?
* Datatypes –
  + **AR: Jeff Hammond to provide a strawman of the data types to be supported**
* MPI\_Probe, MPI\_Mprobe –
  + **AR: Need to complete the man page**
* Tag matching
  + **AR: Figure out how to handle the many-to-one problem of re-use of receive buffer resources – note: this is not restricted to tag matching**
* Flow control –
  + **AR: requires further discussion**.
* Tool interface hooks – should there be hooks for setting get/set options other than in header files? There is interest from MPI tool writers, but probably not for release 1.0. Does this necessarily have to be included in libfabrics, or is it in a separate library?
* Completions/Counters – is there value in defining other counters than pure completion counters such as remote counters. No specific use case has been identified yet.

MPI Initialization group – Main focus was on GetInfo

* Separate discovery from actual resource reservations
  + **AR: Resolve AV map versus AV index issues**
  + **AR: Resolve shared memory – how to share the address table in the same address space**
  + **AR: Need a thread-safe way of multiplexing streams across a single connection**
  + **AR: Address compression?**

SHMEM/one-sided working group

Word doc uploaded to ofa website: <https://www.openfabrics.org/downloads/OFIWG/Hillsboro%20F2F%202014-0819/SHMEM_one-sided_working%20group_2014_0819.docx>

Goals for Illustrative Examples

1) setup/initialization

2) data transfer (put/get) with bulk (fence/quiet) synchronization

3) bundled put operations (counting puts)

4) put with notification

5) put/get/amo types - blocking / nonblocking implicit / nonblocking explicit

6) atomic operations aka AMOs

Beyond SHMEM (or not common SHMEM)

1) General active messages

2) Noncontiguous transfers (strided, vector, subarray, etc.)

3) integer / floating-point accumulate (atomics or active messages?)

Endpoint type - for a simple pseudo code (and for scalability) FID\_RDM is best

* endpoint ops FI\_INJECT (inline in ib speak),
* use fi\_setopt/fi\_getopt to query set FI\_OPT\_MAX\_INJECTED\_SEND
* FI\_REMOTE\_COMPLETE
* must have FI\_WRITE\_COHERENT
* **Open question: How do we support transports like Mellanox DCT ?**

Completion notification - use EQ (not counter for now, may use fid\_ctr for bundled ops)

* use FI\_EVENT to control EQ generation (for blocking and non-blocking explicit)
* use fi\_sync for shmem\_quiet/shmem\_fence/MPI\_Win\_flush(\_local) and non-blocking implicit.
* **Note: fl\_sync does not map well on shmem\_fence().**
* can use the inject option for small puts/put style AMOs

For hints in fi\_getinfo we need these for ep\_cap

* ep\_cap = FI\_RMA | FI\_ATOMICS | FI\_INJECT
* may use FI\_MSG for internal control messages
* op\_flags = FI\_REMOTE\_COMPLETE
* domain\_cap = FI\_WRITE\_COHERENT
* would also like FI\_USER\_MR\_KEY | FI\_DYNAMIC\_MR
* type = FI\_RDM (for this simple example)
* addr\_format = FI\_ADDR\_INDEX
* endpoint attributes
* inject size (get it) we don’t want this to be big, only 8 bytes or so
* check the max\_order\_xxx\_size to see if non-zero
* msg\_order (don’t need order for our example)

**Do need a way to get SHMEM thread hot. How do we do this?  Multiple endpoints? Or multiple domains?**

/\*

 \* lawyer blurb goes here

\*/

struct fi\_info\_shmem, \*fi\_info\_out=NULL;

struct fid\_fabric \*fid\_fabric\_shmem;

struct fid\_domain \*fid\_domain\_shmem;

struct fid\_ep \*fid\_ep\_shmem;

struct fid\_eq \*fid\_eq\_shmem;

struct fi\_eq\_attr eq\_attr\_shmem;

void start\_of\_shmem\_init(void)

{

/\* see all the info stuff above to see how we initialize \*/

fi\_getinfo(“fast-local-rdma-dev-ip-addr”, NULL,0UL,&fi\_info\_shmem,&fi\_info\_out);

assert (fi\_info\_out != NULL);

/\* assume we only got one fi\_info\_out back, i.e. don’t care about multiple rails right now\*/

fi\_fabric(fi\_info\_shmem->fabric\_name, 0, &fid\_fabric\_shmem, (void \*)our\_shmem\_context);

fi\_fdomain(fid\_fabric\_shmem, fi\_info\_shmem, &fid\_domain\_shmem, NULL, (void \*)our\_shmem\_context);

/\* we skip binding EQ to the domain \*/

/\* open the end point \*/

fi\_endpoint(fid\_domain\_shmem, fi\_info\_shmem, &fid\_ep\_shmem, (void \*)our\_shmem\_context);

memset(&eq\_attr\_shmem,0,sizeof(eq\_attr\_shmem));

eq\_attr\_shmem.domain =  FI\_EQ\_DOMAIN\_COMP;

eq\_attr\_shmem.mask = FI\_EQ\_ATTR\_MASK\_V1;

eq\_attr\_shmem.format = FI\_EQ\_FORMAT\_CONTEXT;

/\* open the eq \*/

fi\_eq\_open(fid\_domain\_shmem, &eq\_attr\_shmem, &fid\_eq\_shmem, (void \*)our\_shmem\_context);

/\* bind the eq to the ep, we only want entries back when we ask for them \*/

fi\_ep\_bind(fid\_ep\_shmem, fid\_eq\_shmem, FI\_EVENT);

/\* now set up the adress vector \*/

/\* use out of band mechanism to exchange ip addrs for our node \*/

Rsockets group – key topics and issues

* Memory registration.
* Buffering: everything in sockets is inherently buffered.
* Polling model w.r.t. sockets is completely wrong compared to libfabrics.
* How to address?
  + If you really want these semantics, stick with IPoIB
* Fork support – sockets model you cannot assume that you do forking early on.
  + **AR: define what is needed from fork, and**
  + **AR: what changes are needed to the API to support fork?**
* Immediate data size – currently libfabrics is 64 bits, there have been requests for even larger.
  + Expose 63 bit immediate data above the API.
  + Provide a simulation (do 128 immediates as 64 bits under the covers
  + Leave it to the provider to provide native support for 128 bits.
  + **AR: Evaluate these three options**