Hewlett Packard Enterprise

Persistent Memory and HPC
Enabling New Programming Paradigms

Dave Emberson
Distinguished Technologist
HPC Advanced Technology, Exascale, and Federal Programs
emberson@hpe.com

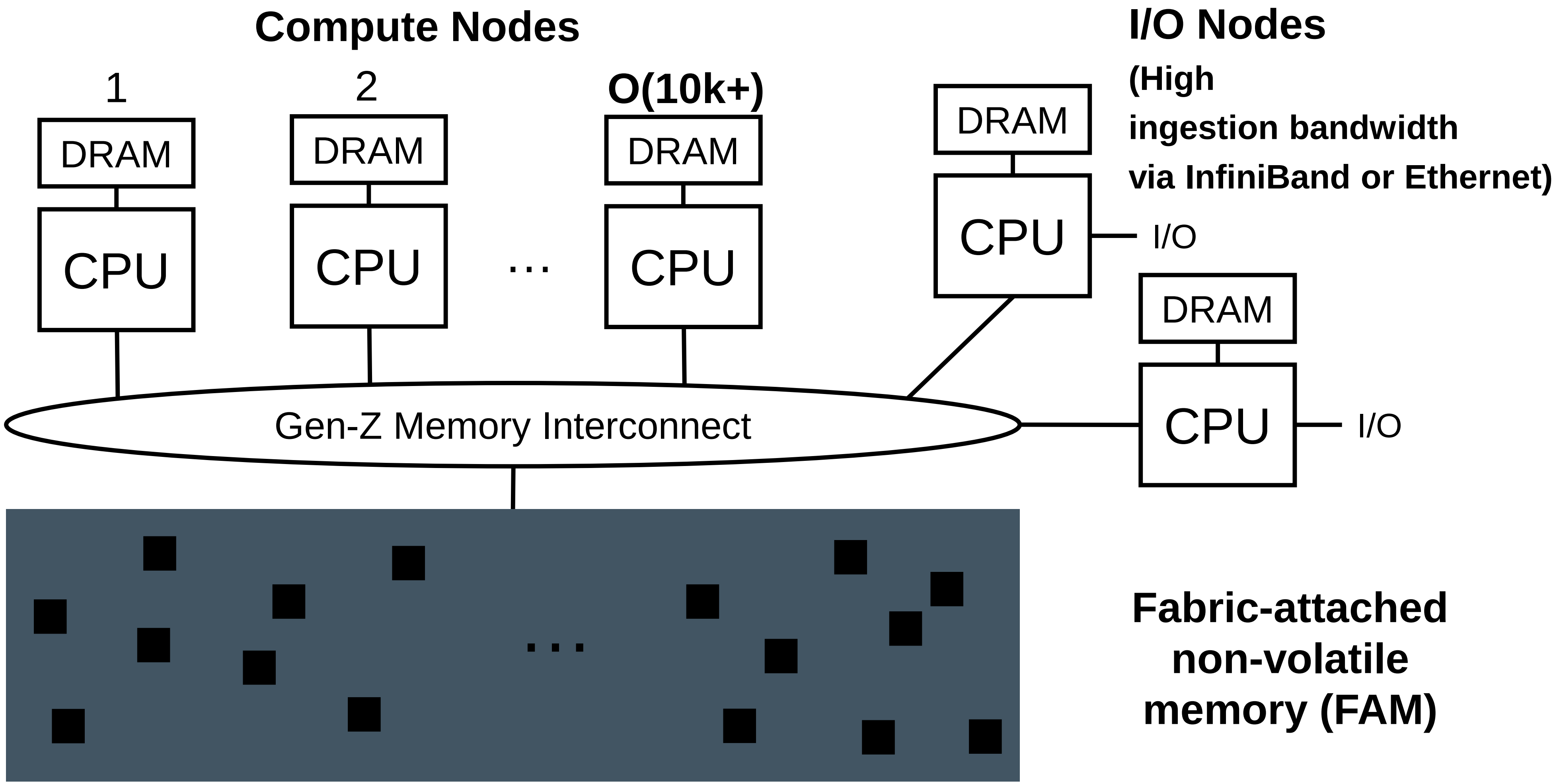
Flash Memory Summit
August 8, 2018

Memory-Driven HPC Architecture

Memory-Driven HPC Design Study

- Define a notional system architecture
  - High performance Gen-Z memory-semantic fabric
  - Extreme scale and capacity
  - Processor and GPU agnostic
  - O(10s of PiBs) of highly resilient fabric-attached memory (FAM)
    - Byte addressable, non-volatile
- Comprehensive software stack definition
  - Seamless convergence of HPC and Cloud workloads (traditional HPC, data analytics, AI)
  - New APIs for FAM access and resilient runtime
- Perform in depth application-specific performance modeling

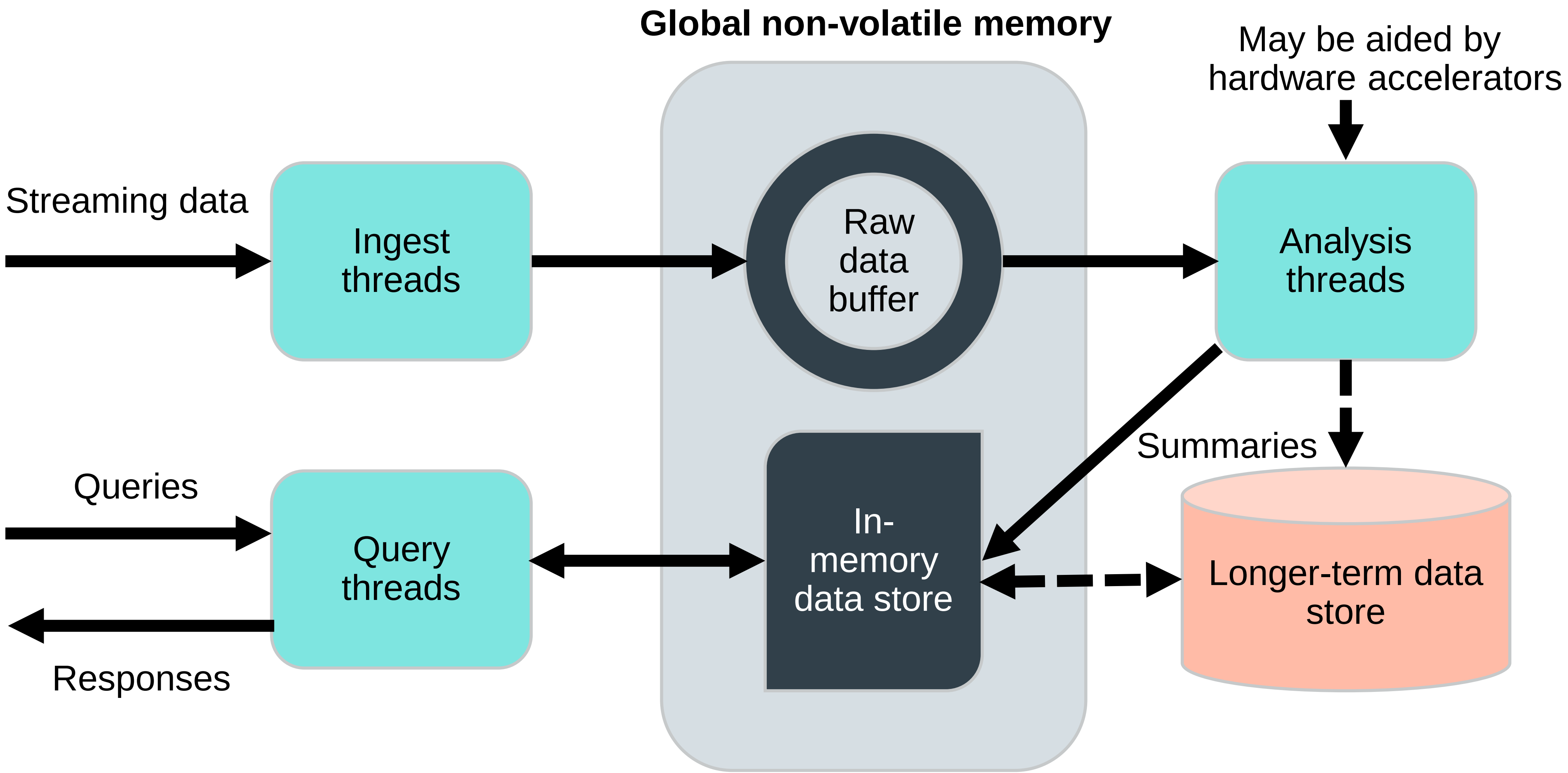
Memory-Driven HPC Architecture



New Programming Paradigms for Memory Driven HPC

Hewlett Packard Enterprise

Idealized Workflow for HPC and Data Analysis



Non-Volatile Fabric-Attached Memory Enables New Possibilities

- Simplified programming model: OpenFAM proposal
  - Globally accessible shared data structures in FAM visible to all participating compute threads
  - Efficient one-sided data access; pass pointers → reduced message passing overhead
- New runtime model: Task model with work-oriented synchronization
  - Calling task spawns workers; blocks until work is completed (traditional PGAS barriers block PEs until other PEs reach the synchronization point)
  - Better load balancing and robust performance for skewed and variable workloads; processes are equally able to service requests and analyze any part of the dataset
  - Simplified coordination: processes don't need to exchange messages to establish common view of global state
  - State is maintained in highly resilient FAM; compute nodes and FAM fail independently, so persistent state in FAM will survive failures of processes or compute nodes
  - Any other worker can pick up where the failed worker left off
  - Checkpointing is no longer necessary

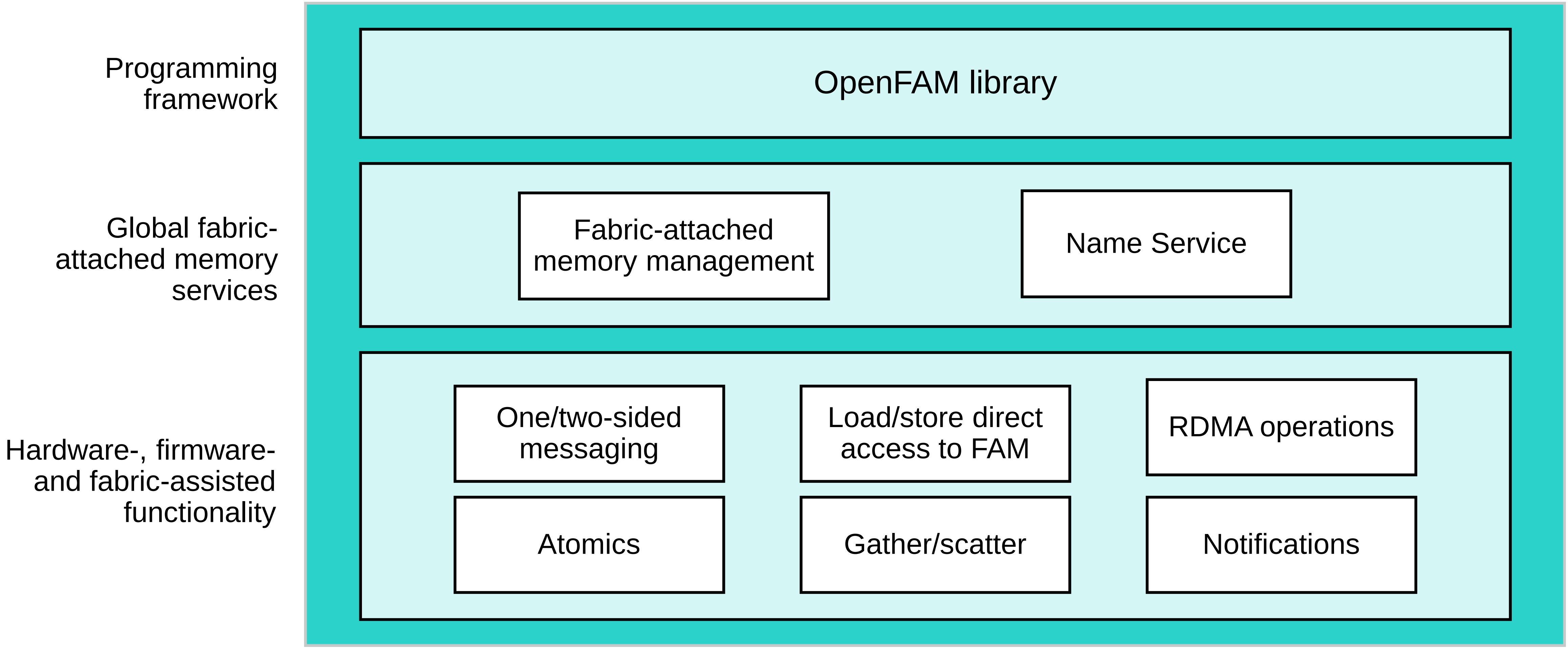
OpenFAM API

Kim Keeton, Sharad Singhal
kimberly.keeton@hpe.com,
sharad.singhal@hpe.com

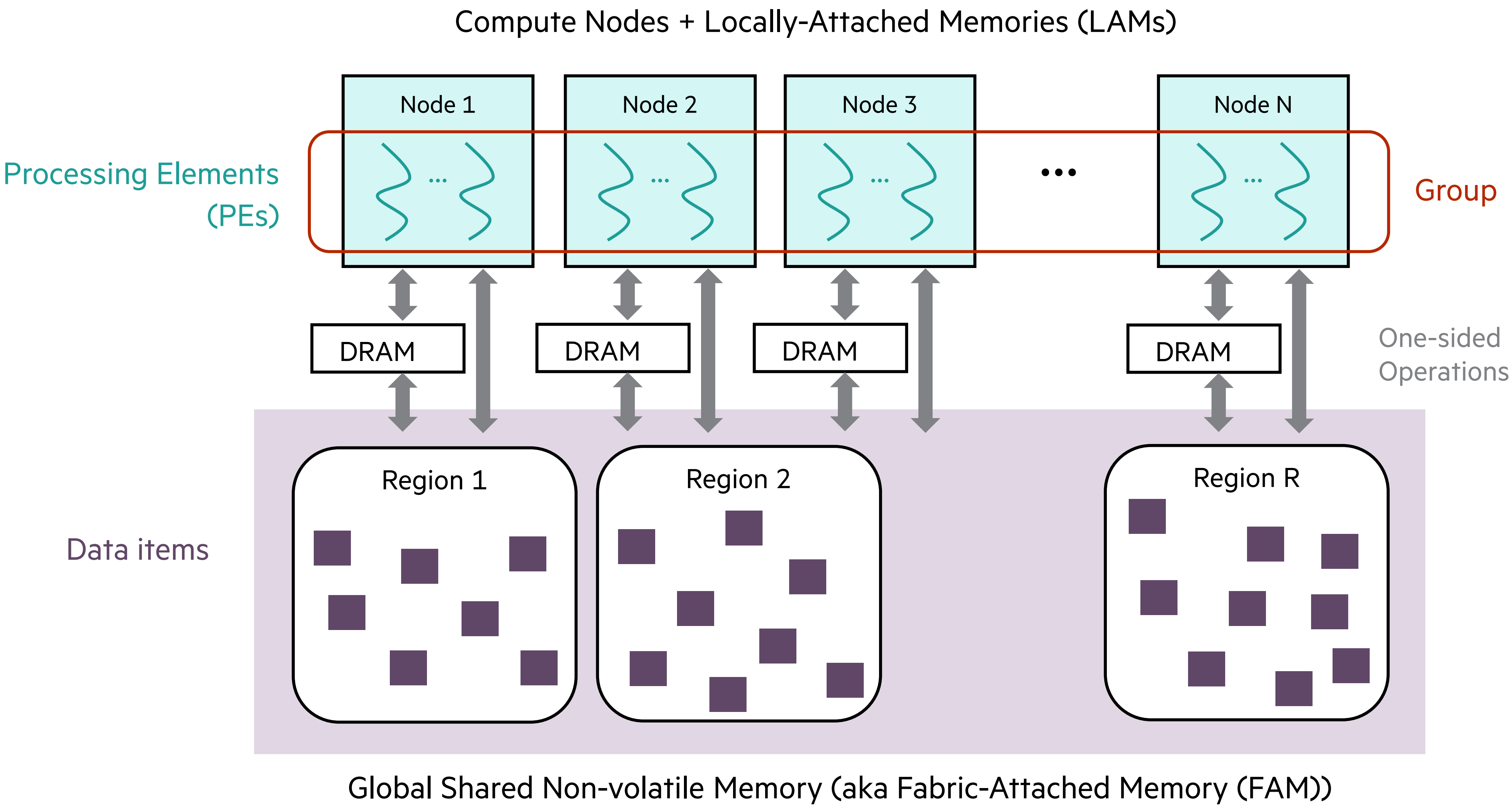
OpenFAM: Programming API for Fabric-Attached Memory

- Inspired by OpenSHMEM (http://openshmem.org): open source partitioned global address space (PGAS) library with one-sided communication, atomic and collective operations
- Used to access/manage persistent fabric-attached memory (FAM)
- FAM is persistent; data can live beyond program invocation.
- One-sided/unmediated access to fabric-attached memory

OpenFAM software stack



OpenFAM concepts



Regions vs. data items

- Regions permit definition of sections of FAM with different characteristics to accommodate different data needs.
- Useful to permit multiple regions associated with a given job to accommodate different data needs. Examples:
  - No redundancy for communication or scratch space
  - Redundancy for computation results
- Named regions of FAM enable sharing between PEs of a given job and also between jobs (for persistent data)
- Region forms basis for heap allocator in memory management routines
  - Data items are allocated using heap allocator

Descriptors

– Descriptors are opaque read-only data structures that uniquely identify a location in FAM and permissions required to access that location

– Descriptors are portable across OS instances

  – Use base + offset addressing

  – Can be freely copied and shared across processing nodes by the program

```
typedef struct {
    int accessPermissions;  // flags indicating access permissions
    long regionId;    // region ID for this descriptor
    size_t offset;    // offset w/in region for start of descriptor’s memory
    size_t size; // size (in bytes) of memory associated with descriptor
} Fam_Descriptor;
```

```
typedef struct {
    Fam_Descriptor descriptor; // descriptor pointing to memory region
    Fam_RedundancyLevel redundancyLevel;
    // redundancy options for this region
    // futures: additional parameters, such as quality of service
} Fam_RegionDescriptor;
```

API classes of interest

- Initialization
- Query
- Allocation
- Data path
- Atomics
- Memory ordering
- Collectives (barriers)

Initialization APIs

- shmem_init: collective to allocate and initializes OpenSHMEM library resources
- shmem_my_pe: returns number of calling PE
- shmem_n_pes: returns number of PEs for a program
- shmem_finalize: collective to release OpenSHMEM library resources. Only terminates the OpenSHMEM part of program, not entire program.
- shmem_global_exit: routine that allows any PE to force termination of entire program
- shmem_ptr: returns pointer to data object on specified PE (permits ordinary ld/st access)

- int fam_initialize(Fam_Options *options): allows worker PE to join a group at job initialization or on demand
  - Creates/locates coordination data structures in FAM
  - Adds info for this process executable to those structures
  - Open question: access control mechanism
    - Default: Unix style user/group/other
    - Also possible: PKI: private/public key pairs or access tokens
  - Open question: what additional options are required/desired? (see data management slide for region creation)
- void fam_finalize(char *group): disconnects the PE from the app. Only terminates the OpenFAM part of the program.
- void fam_exit(int status): allows any PE to force termination of the entire program.

Query APIs

- shmem_my_pe: returns the number of the calling PE
- shmem_n_pes: returns number of PEs running in a program

- char **fam_listOptions(void): lists known options for the FAM library
- const void* fam_getOption(char *optionName): query FAM library for an option
- void fam_setOption(char *optionName, void *option): set a name -> option mapping. Options can be of arbitrary type.
- void fam_register(char *name, Fam_Descriptor *descriptor): register mapping of name -> data itemFAM descriptor with name service.
  - Assumptions: a name is unique within its region, and a descriptor may be associated with multiple names
  - Note: region names are automatically registered
- void fam_unregister(char *name, char *regionName): unregister name -> FAM descriptor mapping for data item in region regionName
- Fam_Descriptor *fam_lookup(char *itemName, char *regionName): look up data item by name
- Fam_RegionDescriptor *fam_lookupRegion(char *name): look up region by name

Allocation APIs (region management)

Region APIs: manage creation, destruction of regions

- Fam_RegionDescriptor = fam_createRegion(char *name, long size, int permissions, Fam_RedundancyLevel level, …): allocates region of size bytes in FAM, with associated options
  - Region can be further allocated through heap management APIs (see next slide). One heap allocator per region.
  - Regions are long-lived and automatically registered with name service
  - System may impose system-wide or user-dependent limits on individual and total region allocations
- void fam_destroyRegion(Fam_RegionDescriptor *descriptor): destroys the region
  - Employs appropriate delayed reclamation to accommodate ongoing users

Allocation APIs (data item / heap management)

SHMEM's symmetric heap management APIs

- Notes: all routines call shmem_barrier_all before returning to ensure all PEs participate in memory allocation. User must call routines with identical argument(s) on all PEs.
- shmem_malloc: return pointer to block allocated from shared symmetric heap
  - void *shmem_malloc(size_t size)
- shmem_free: deallocate block associated with ptr
  - void shmem_free(void *ptr)
- shmem_realloc: change size of ptr's block to size
  - void *shmem_realloc(void *ptr, size_t size)
- shmem_align: returns pointer to aligned block allocated from shared symmetric heap
  - void *shmem_align(size_t alignment, size_t size)

FAM heap allocator APIs: manage data item allocation from region

- Fam_Descriptor *fam_allocate(char *name, size_t nbytes, int permissions, Fam_RegionDescriptor *region): allocates space within a region
- void fam_deallocate(Fam_Descriptor *descriptor): used by PE to indicate that it's done with allocation associated with descriptor
  - Note: expect that this will trigger delayed reclamation, in case another PE is accessing descriptor, or until it is more optimal for reclamation pass
- void fam_resizeRegion(Fam_RegionDescriptor *descriptor, size_t nbytes): change size of region allocation
  - Note: shrinking size of region may make descriptors to data items within the region invalid.
- void fam_changePermissions(Fam_Descriptor *descriptor, int permissions): change permissions associated with a descriptor

Data path APIs: get / put

SHMEM blocking:

- void shmem_put(TYPE *dest, const TYPE *source, size_t nelems, int pe): blocking write to remote PE's memory
  - shmem_p puts a single element
  - Returns after data is copied out of source array. Two successive puts may deliver data out of order unless shmem_fence is used.
- void shmem_get(TYPE *dest, const TYPE *source, size_t nelems, int pe): blocking read from remote PE's memory
  - shmem_g gets a single element

Non-blocking:

- void shmem_put_nbi(TYPE *dest, const TYPE *source, size_t nelems, int pe): non-blocking write to remote PE's memory
- void shmem_get_nbi(TYPE *dest, const TYPE *source, size_t nelems, int pe): non-blocking read from remote PE's memory
- Note: non-blocking calls require shmem_quiet to ensure completion; may arrive out of order

Note: these operations copy data between FAM and local memory

- void fam_put(void *local, Fam_Descriptor *descriptor, size_t offset, size_t nbytes): write nbytes from PE's local memory to FAM descriptor (+ offset)
  - Assumption: fam_put is non-blocking, with host bridge returning completion of operation.
- void fam_get(Fam_Descriptor *descriptor, void *local, size_t offset, size_t nbytes): read nbytes from FAM descriptor (+ offset) to PE's local memory
  - Assumption: fam_get is blocking.
- Notes/questions:
  - If needed, in the future we can extend the API to provide both blocking and non-blocking calls for both put and get.

Data path APIs: scatter/gather accesses

- shmem_iput: copies strided data to specified PE
  - void shmem_iput(TYPE *dest, const TYPE *source, ptrdiff_t dstride, ptrdiff_t sstride, size_t nelems, int pe)

- shmem_iget: copies strided data from specified PE
  - void shmem_iget(TYPE *dest, const TYPE *source, ptrdiff_t dstride, ptrdiff_t sstride, size_t nelems, int pe)

Constant stride

- void fam_scatter(void *local, Fam_Descriptor *descriptor, long firstItem, long nitems, long stride, size_t nbytes): copies data from contiguous structure in local PE memory to strided locations within FAM. Copies nitems of length nbytes each to offsets starting at firstItem with stride.

- void fam_gather(Fam_Descriptor *descriptor, void *local, long firstItem, long nitems, long stride, size_t nbytes): copies data from strided locations within FAM to a contiguous structure in local PE memory. Copies nitems of length nbytes each from offsets starting at firstItem with stride.

Indexed

- void fam_scatter(void *local, Fam_Descriptor *descriptor, long nitems, long *itemIndex, size_t nbytes): copies data from contiguous structure in local PE memory to non-contiguous locations within FAM. Copies nitems of length nbytes each to indexes specified in itemIndex.

- void fam_gather(Fam_Descriptor *descriptor, void *local, long nitems, long *itemIndex, size_t nbytes): copies data from non-contiguous locations within FAM to a contiguous structure in local PE memory. Copies nitems of length nbytes each from indexes specified in itemIndex.

Data path APIs: direct access (map/unmap)

Note: these operations permit subsequent direct load/store access to fabric-attached memory.

– void *fam_map(Fam_Descriptor *descriptor): maps a data item from FAM into the PE's address space

– void fam_unmap(void *local, size_t nbytes): unmaps a data item from the PE's address space

Hewlett Packard Enterprise

Atomics APIs

- SHMEM fetching routines: return original value and optionally update remote data in single atomic operation. Return after data has been delivered to local PE.
  - shmem_fetch: atomically fetches value of remote data object
  - shmem_swap: atomic swap to remote data object
  - shmem_cswap: atomic conditional swap on remote data object
  - shmem_finc: atomic fetch-and-increment on remote data object
  - shmem_fadd: atomic fetch-and-add on remote data object
- SHMEM non-fetching routines: update remote memory in single atomic operation. Non-blocking: routine starts the atomic operation and may return before execution on remote PE. Need shmem_{quiet, barrier, barrier_all} to force completion.
  - shmem_set
  - shmem_inc: atomic increment on remote data object
  - shmem_add: atomic add on remote symmetric data object

RDMA operations

- OpenFAM fetching routines:
  - 32b and 64b integer: fetch, swap, compare-and-swap, add, subtract, min, max, and, or, xor
  - Unsigned 32b and 64b integer: compare-and-swap, add, subtract, min, max
  - 128b integer: compare-and-swap
  - Float/double: add, subtract, min, max
- OpenFAM non-fetching routines:
  - 32b and 64b integer: add, subtract, min, max, and, or, xor
  - Unsigned 32b and 64b integer: add, subtract, min, max
  - Float/double: add, subtract, min, max

Collectives APIs

- Note: all collectives are blocking and return on completion
- shmem_barrier_all: registers PE arrival at barrier. Suspends PE execution until all other PEs arrive at barrier and all local and remote memory updates are completed.
- shmem_barrier: same as shmem_barrier_all, but with respect to subset of PEs
- shmem_broadcast
- shmem_collect, shmem_fcollect
- shmem_alltoall, shmem_alltoalls
- Reduction operations
  - And, max, min, sum, prod, or, xor
- shmem_wait: waits for a variable on the local PE to change (after update by remote PE)

- void fam_barrier(char *group): registers a PE's arrival at a barrier, and suspends PE execution until all other Pes arrive at barrier.
  - As an initial step, we assume a barrier that implements semantics similar to shmem_barrier_all.
- Notes on desired barrier semantics:
  - SHMEM defines barriers in terms of a fixed set of PEs reaching a particular point, and doesn't tolerate failures
  - For resilience, we want to redefine barrier to be in terms of completed work (regardless of which PEs complete the work)
    - No failures: equivalent to shmem_barrier_all
    - Failures: runtime system needs to reallocate work for failed PE

Memory ordering APIs

- shmem_quiet: waits for completion of all outstanding put, atomics, memory store and non-blocking put and get routines to symmetric data objects issued by PE to any/all remote PEs
  - void shmem_quiet(void)
- shmem_fence: assures delivery order of put, atomics, and memory store routines to symmetric data objects issued by PE to a particular target PE
  - void shmem_fence(void)
- Basic interpretation: all operations before shmem_quiet/fence must complete before any operations after shmem_quiet/fence

- void fam_fence(void): waits for all outstanding memory operations between PE's local memory and FAM to complete
- Notes:
  - It can be used to enforce ordering of outstanding FAM operations from local memory
  - Fence/quiet distinction between a single target PE vs. all target PEs probably doesn't make sense here, unless we want to call out individual memory controllers.
  - This has the semantics of shmem_quiet. We call it fence rather than quiet, to be more consistent with mfence/sfence.

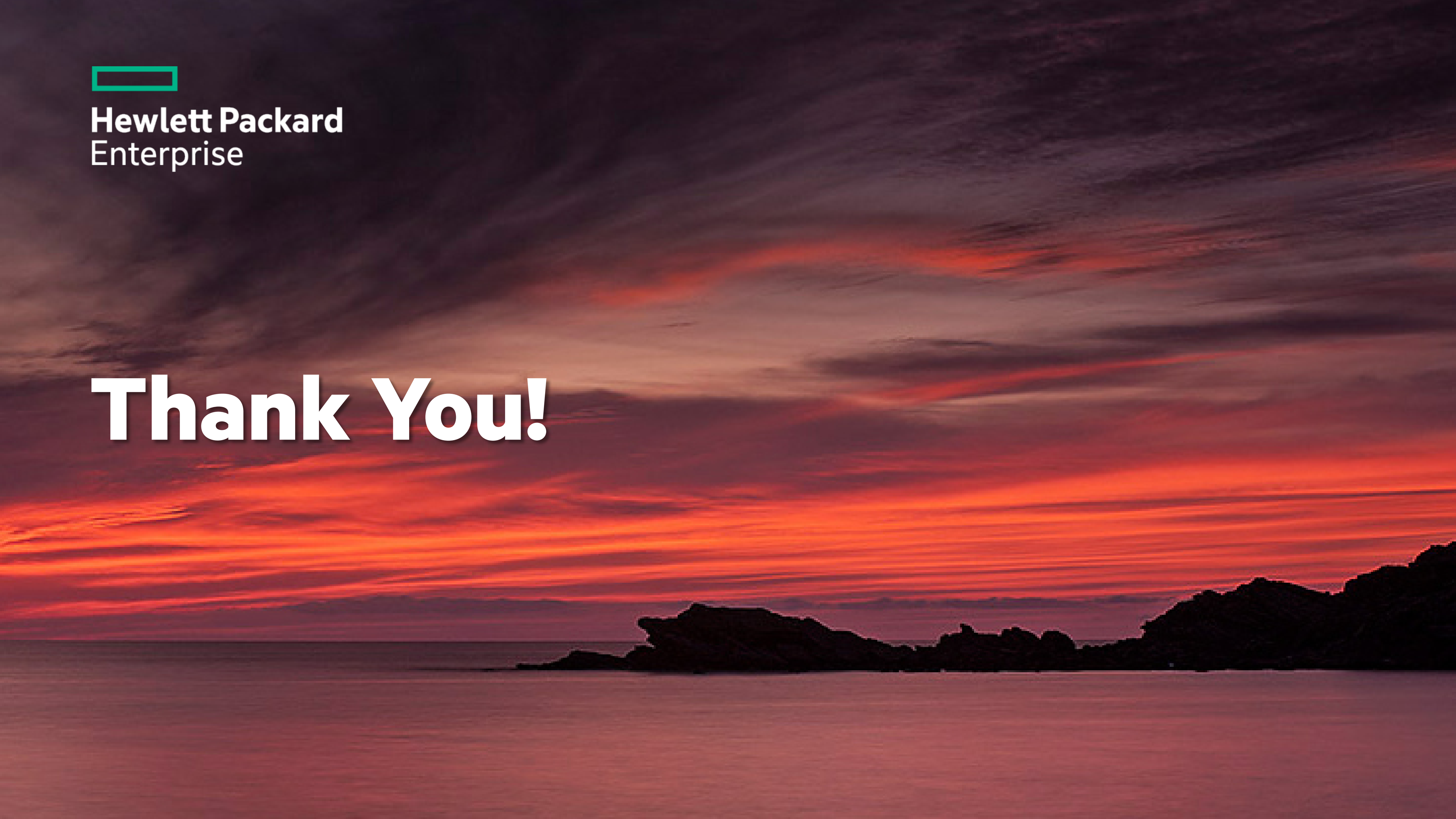
OpenFAM Status

– Some sample applications "ported" and running in simulation

– API defined and presented to the OpenSHMEM Steering Committee. OSC has created a Memory Model subcommittee to study adopting OpenFAM concepts in OpenSHMEM 2.0 scheduled for 2020 release.

– Draft API specification released on github at https://github.com/OpenFAM/API and open for public comment

– Comments be addressed to Kim Keeton (kimberly.keeton@hpe.com) or Sharad Singhal (sharad.singhal@hpe.com)


Hewlett Packard
Enterprise
Thank You!