UCCS: A Communication Substrate for Open SHMEM (and more)

ICL “Friday Lunch”, Oct. 31 2014

Aurélien Bouteiller

(with work from Chunyan Tang, Thomas Hérault, George Bosilca)
What is Open SHMEM

- Symmetrical Hierarchical MEMory
- SHMEM is a 1-sided communication library
  - C and Fortran PGAS programming model
  - Point-to-point and collective routines
  - Synchronizations
  - Atomic operations
- Can take advantage of hardware offload
  - Performance benefits

1-Sided: Put, get (the target is passive)

vs

2-Sided: MPI_SEND/MPI_RECV (the target matches the calls)
Partitioned Global Address Space

- **Shared memory** (ex. OpenMP, threads): global (unpartitionned) address space, all PEs (processing elements) can access all memory (implicit communications)

- **Distributed memory** (ex. MPI): partitioned address space, PEs have private memory and can only exchange messages

- **PGAS** (ex. Open SHMEM): has both private and shared data, explicit communications in a symmetric address space
Symmetric Partitioned Memory

- Shared variables have the same address from all PEs
- Access to remote shared variables using RMA operations
- Thanks to symmetric memory, operands to put/get are computed as normal addresses

```c
int main (void) {
    int *x;
    ...
    start_pes(0);
    ...
    x = (int*) shmalloc(sizeof(x));
    ...
    ...
    shmem_barrier_all();
    ...
    shfree(x);
    return 0;
}
```
Communication operations

OpenSHMEM

/*Symmetric work array*/
int number;

/*Initialize the OpenSHMEM library*/
start_pes(0);

/* Get the number of PEs*/
int npes = num_pes();

/*Get the rank of the PE */
int me = my_pe();

if(me == 0) {
    shmem_int_put(&number, &number, 1, 1);
    shmem_quiet();
}
shmem_barrier_all();

if(me == 1)
    printf("PE 1 received number %d\n", number);

MPI

/*Initialize the MPI environment */
MPI_Init(NULL, NULL);

/* Get the number of processes */
int world_size;

MPI_Comm_size(MPI_COMM_WORLD, &world_size);

/*Get the rank of the process */
int world_rank;

MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

if (world_rank == 0) {
    int number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
}
else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 1 received number %d\n", number);
}

Put/Get (with basic stride typing)
Atomic operations (inc, swap, fetch_add, etc.)
Collective operations (bcast, etc.)
Synchronizations (fence/quiet, barrier)
Conversion of MPI applications

• Chunyan works on converting LAMMPS
• Conversion is not straightforward (lack of some concepts, implicit 2-sided synchs that are semantically important must be enforced with shmем_barrier, which is hard to use for subgroups)
• Even with a brutal implementation (with extraneous synchs), shmем version competitive with optimized MPI code

Fig. 2: Weak scaling for test results for LAMMPS

Fig. 3: Strong scaling test results for LAMMPS
UCCS architecture

- Common low-level scalable, robust, portable, simple and performance driven communication API for multiple parallel programming models over modern network interfaces
- Provide common network code for implementing programming models, increasing code reusability and reducing development effort
- Support hybrid programming environments efficiently, mixed programming models
UCCS thread safety

• Multicore processors
  • Want to have the benefit from shared memory, direct access etc
  • MPI+X, also applies to OpenSHMEM +X
  • OpenSHMEM implementation may benefit from multiple internal threads (improves drainage rate etc in some cases)

• We need a thread safe communication substract
  • Thread safety is easy, but can become expensive quickly
  • Must define sensible behavior for all APIs (RMA, Active Messages, Atomics, RTE, etc).
  • We have control on the spec, we can define what we want 😊

Coarse grain locking

Fine grain locking/lockless thread safety
Naïve thread safety

- This is the poor performance with a coarse grain mutex protecting all routines...
- Not very enticing, can we do better?
Thread Safety: RTE API

- UCCS specification provides a portable Runtime Environment API
  - Processor architecture, endianess, hardware locality info database
  - Network ID database
  - I/O forwarding
  - Out of band communication service
- **UCCS Specification:** assume all RTE routines are thread safe
- Can employ PMI, Open RTE, STCI, Slurm, etc. as the backend
- Unfortunately not all of these RTE backends are thread safe
- Used mostly when establishing new connections: RTE is not performance critical
- Implementation delegates the RTE in a separate thread. All commands delegated to the RTE thread

```
<table>
<thead>
<tr>
<th>Command queue</th>
<th>Result queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>rte_send_nb</td>
<td>rte_send_nb</td>
</tr>
<tr>
<td></td>
<td>rte_wait</td>
</tr>
</tbody>
</table>
```

App. threads
RTE thread
Thread safety: Posting Operations

- **uccs_put_large_contiguous_nb**(  *ctx, ep, ..., desc* )
  - Creates a descriptor for the operation (aka a request)
  - Can be completed later by uccs_wait, uccs_test

- In the UCCS specification, all postings are thread safe (can be issued simultaneously from multiple threads, on the same communication channel)

- Implementation of most RMA operations uses fine grain locking (only active message posting is not yet), using OPAL lockless lists and data structures

- There remain some state sharing between threads (due to sharing the endpoint handle)
  - Current proposal in the OpenSHMEM community would help addressing these issues (endpoints, contexts)
Thread Safety: Progress

- UCCS specification: no guaranteed* progress if no thread is calling a progress function
  - uccs_wait, uccs_test, uccs_progress
  - To ensure the semantic of OpenSHMEM asynchronous progress, one thread can be delegated to progress UCCS
  - *The UCCS implementation may spawn threads of its own to progress RMA

- Multiple threads can enter progress simultaneously

- Active Message callbacks may be invoked from any thread that enters progress
  - The callbacks must ensure their own exclusive access to shared variables if needed

- 2 types of AM callbacks can be registered
  - Network_Protocol
    - high priority functions, that demand no additional resource and only a few computations. Can run on any thread.
  - User
    - for more demanding operations. Can run on User Threads only
Performance Evaluation

- Infiniband 20G cluster
  - 2x Intel Xeon E5520 (Nehalem) 4cores (total 8 cores/node)
  - Linux CentOS 6.5

- Intel MPI Benchmarks 4.0 (RMA)
  - Open MPI 1.7.5 (non thread safe build)
  - Deploying $t$ processes per node (single threaded), each process pinned to a physical core (multi mode)
  - Put/Get bidirectional aggregate mode benchmarks are considered

- UCCS benchmarks
  - UCCS version (thread safe extended)
  - Deploying 1 process per node (with $t$ threads), each thread pinned to a physical core
  - Similar communication pattern to the IMB deployment
1050 AM message sent from the origin, T threads are spinning in uccs_progress()

When the 1050 messages have been received, an ack is sent back

Some congestion is observed (some coarse grain locks still present in the implementation)

In all threaded cases, injection rate improved with MT compared to single thread
UCCS Fairness

- Same AM benchmark, 8 receive threads
- Perfect load balance: ~131 callbacks per threads
- No thread is favored, good spread
- Standard deviation between runs is low
- Is this a good thing? (think interrupt spread)
The RMA benchmarks

- MPI deployments use 1 process per core
- UCCS deployments use 1 thread per core
The RMA benchmarks

- MPI deployments use 1 process per core
- UCCS deployments use 1 thread per core

MPI processes pairs, even ranks on first node, odd ranks on second node
Multithreaded Put benchmark

- **1 thread:**
  - UCCS bandwidth slightly better than MPI

- **2 threads:**
  - UCCS and MPI bandwidth improved (especially for medium messages)

- **More threads:**
  - MPI reaches best bandwidth for 4 threads, congestion starts to happen for 8 threads
  - UCCS also congested for 4/8 threads
Multithreaded Get benchmark

- Same observation hold for GET benchmark
Concluding Remarks

• Evaluated porting difficulty/performance for OpenSHMEM LAMMPS
• Designed thread safety interface to UCCS specification
• Early performance results encouraging
  • Competes with multiple MPI processes (in a non-thread safe MPI build), with multiple non-shared endpoints
  • Observed good overall fairness, even when contentions are observed
• Future works
  • Observing behavior with multiple channels/qpairs to separate threads
  • Observing overlap in synthetic benchmarks
  • Observing OpenSHMEM application performance
  • Investigating “endpoints/PE-contexts” proposed extension to OpenSHMEM
  • Investigating “mixed model” programming (OpenSHMEM+MPI, OpenSHMEM+dataflow) when sharing the base communication substruct