Implementing Split-Mode Barriers in OpenSHMEM

Michael A Raymond
SGI MPT Team Leader

sgi

Agenda

- SHMEM barriers
- Dynamic Sparse Data Exchange Problem
- Design Evolutions
- Results
- Conclusions

Barriers in OpenSHMEM

- shmem_barrier_all()
- shmem_barrier() for sub-groups
- Requires a symmetric sync variable
- All prior RMAs done by barrier completion

Split-mode Barriers

- Split-mode vs asynchronous barriers
- barrier_post(), barrier_test(), barrier_wait()
- Allow PEs to get other work done while waiting for everyone to have posted
- Recently added in MPI 3.0

Dynamic Sparse Data Exchange

- Data exchange problem with shifting graph links
- Reviewed by Hoefler et al.
- They proposed a microbenchmark to compare alternative solutions
- Fundamentally a message passing model

Their Microbenchark

```
for (1 – 1000) {
    randomly choose log(#PEs)
    generate 1-1024 bytes of data for them
    send the data out
    discover our incoming peers
    get their data
}
```

Data Structures

- Set of buffers in the symheap
- Message descriptor data type
  - PE, buffer location, buffer size
- Queue of descriptors
- Queue pointer atomic variable

OpenSHMEM Implementation

```
for (1 – 1000) {
    generate_peers_and_data();
    send_notices(); /* FADD + Put */
    barrier_start();
    do {
        done = barrier_test();
        while (probe for notices) shmem_getmem();
    } while (!done);
    shmem_barrier_all(); /* Gets done? */
}
```

Test Configuration

- SGI ICE-X
- 32 nodes
- Two 12-core Intel Ivy Bridges per node
- Mellanox ConnectX-3 FDR
- 768 PEs

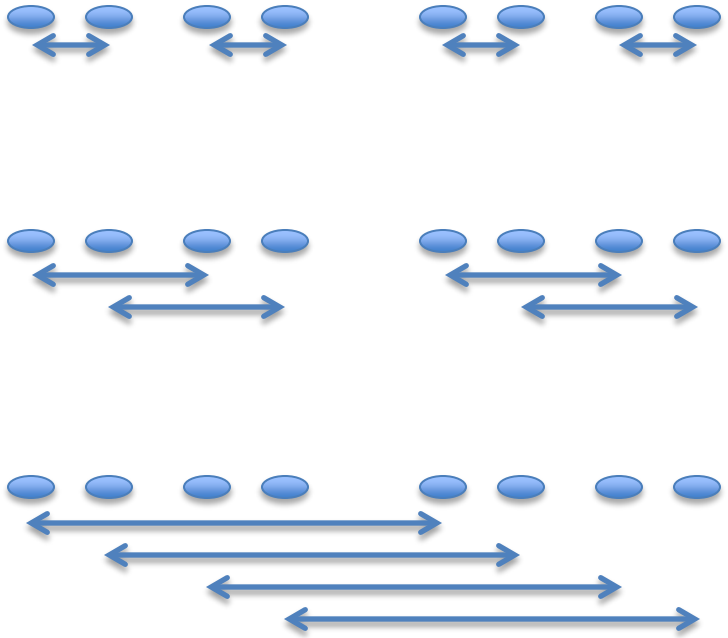
Design Evolutions

Barrier Based on MPI_Ibarrier()

- Based on MPICH 3.0 Ibarrier design
- Uses message passing
- Test code uses standard MPI_Test(), MPI_Wait()
- Results: 0.551

Barriers in SGI SHMEM

- Dissemination barrier
- One 32-bit counter for each exchange
- Using AMOs on the counters allow reuse
- Results: 1.463s



Initial Split-Mode Version

```
barrier_test()
{
        if (first) {
                shmem_int_inc(&words[step], tgt_pe);
                first = 0;
        }
        do {
                if (0 == words[step]) {return 0;}
                step++; tgt_pe = next_partner();
                if (not last step) {
                        shmem_int_inc(&word[step], tgt_pe);
                } else {break;}
        } while (1);
        for (0 .. #steps) {shmem_int_add(&words[i], -1, my_pe);}
        return 1;
}
```

64-bit Integers

- Mellanox IB doesn't do CPU coherent AMOs
- Force all AMOs through the HCA
- 32-bit AMOs require special tricks
- Switched to 64-bit data structure
- Ignored implications for pSync size
- Result: 0.663 seconds

Non-blocking AMO

- Implementation doesn’t need the result of any AMO
- shmem_long_inc() implemented as shmem_long_fadd(,1,)
- The later expects a local result, the former doesn’t
- Altered implementation to not wait for result
- Result: 0.688s, slower

Multiple Outstanding AMOs

- Previous approach allowed a single asynchronous AMO
- What about using more?
- Verified that safe within this specific benchmark
- Up to 2 asynchronous AMOs: 0.678s
- Up to 4: 0.671s
- Both still slower

Barriers without AMOs

- Exhausted AMO approach
- AMOs were only used to save space
- Switched to Puts
- Required multi-phase datastructure
- Result: 0.485 seconds

More Frequent Progress Checks

- Ibarrier() benefits from progress engine inclusion
- Hacked progress engine to poll & update barrier state
- Result: 0.484 seconds, no change
- Might benefit other codes
- “Communicator”-based SHMEM collectives might make this easier

Summary Performance

Experiment	Execution Time
MPI_Ibarrier()	0.551
32-bit AMOs	1.463
64-bit AMOs	0.663
Non-blocking 64-bit AMO	0.688
2 asynchronous 64-bit AMOs	0.678
4 asynchronous 64-bit AMOs	0.671
64-bit Puts	0.485
Progress engine integration	0.484

Conclusions

- Same overall algorithm but different primitives
- Asynchronous barriers can be useful to OpenSHMEM developers
- Post-Wait vs Test
- Communicator Object