From MPI to OpenSHMEM: Porting LAMMPS

C. Tang, A. Bouteiller, T. Herault, M.G. Venkata, G. Bosilca

OpenSHMEM Workshop 2015, Aug. 5, Annapolis, MD
Motivation

- 1 sided model: potential for large gains (major on injection rate, minor in latency)
- How hard is it to deploy, in practice?
- The community needs some recipes and guidelines
Why LAMMPS

- Large-scale Atomic/Molecular Massively Parallel Simulator
- Widely used in production
  - Solid-state materials (metals, semiconductors)
  - Soft matter (biomolecules, polymers)
  - Etc.
- MPI based application
Why LAMMPS

- Rhodopsin (8x8x8x32k atoms)
- Strong Scaling shows potential for substantial improvement

<table>
<thead>
<tr>
<th>nprocs</th>
<th>MPI%</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>5.5</td>
</tr>
<tr>
<td>512</td>
<td>12.6</td>
</tr>
<tr>
<td>1024</td>
<td>18.1</td>
</tr>
<tr>
<td>2048</td>
<td>25.4</td>
</tr>
<tr>
<td>4096</td>
<td>38.5</td>
</tr>
</tbody>
</table>
MPI Usage in LAMMPS

- **Goal**: hybrid MPI/Shmem application, upgrade to high return routines first
- **Profiling with mpiP**: more than 167 MPI call sites
  - MPI_Send dominates the MPI wait time.
  - Remap_3d() sends 32.45% of the data
    - Initial effort on this operation

@512 processes
2-D decomposition of 3D-FFTs

8x8 process grid

1d-fft fast axis
1d-fft mid axis
1d-fft slow axis

Remap1
Remap2
3D-FFT in LAMMPS – $P(x,y,z,e)$

- Multiple 3D FFTs per iteration, work on separate data structures

3D-FFT_e

- 1d-fft fast axis
- 1d-fft mid axis
- 1d-fft slow axis

- Remap0
- Remap1
- Remap2
- Remap3

3D-FFT_x

- 1d-fft fast axis
- 1d-fft mid axis
- 1d-fft slow axis

- Remap4
- Remap5
- Remap6

3D-FFT_y

- 1d-fft fast axis
- 1d-fft mid axis
- 1d-fft slow axis

- Remap4
- Remap5
- Remap6

3D-FFT_z

- 1d-fft fast axis
- 1d-fft mid axis
- 1d-fft slow axis

- Remap4
- Remap5
- Remap6
MPI and its implicit synchros

Simplified view: every process sends and receive from/to multiple peers
MPI and its implicit synchros

**PE0 - Sender**
- FFT_x
- Pack
- Scratch
- MPI_send
- Eager data
- RDMA Data

**PE1 - Receiver**
- FFT_x
- MPI_Irecv
- Matching
- Scratch
- unPack
- out

2 sided: each side provides the target address

Rendez-vous:
- Scratch is ready

- FFT_y

in

unPack
MPI and its implicit synchros

Implicit synchronization

PE0 - Sender

MPI_send

Eager data

Rendez-vous:

Scratch is ready

RDMA Data

MPI_Wait

UnPack

Scratch

MPI_Irecv

Matching

Memory exposure starts

Memory exposure ends

PE1 - Receiver

FFT_x

FFT_y

FFT_x

In

Pack

Scratch

Out
Conversion: *put* target offsets

- **1sided**: other processes need to understand each others’ memory layout
  - Scratch buffers allocated in the symmetric address space
  - Each PE has a different communication plan
    - PE1 receives 1MB from PE0, PE2’s offset in target scratch is then at \( base+1MB \)
    - PE2 receives 2MB from PE0, PE1’s offset in target scratch is then at \( base+2MB \)
  - The plan is invariant: we exchange all offsets once, during startup

```c
1 ... plan->remote_offset = (int *) shmem_malloc(nprocs*sizeof(int));
2 for( i = 0; i < plan->nrecv; i++)
3    shmem_int_p(&plan->remote_offset[me], plan->recv_bufloc[i],
4                 plan->recv_proc[i]);
5 shmem_fence();
6 ...
```

Listing 1.2: Exchanging the offsets in the target scratch buffers; a parameter to *shmem_put* that was not required with MPI_Send.
Conversion: *put* completion signaling

- In two sided, `MPI_Waitany` tracks the completion of `iRecv`
- In one sided, the target cannot tell directly when `put` has completed
  - Plan->remote_status is initialized in status “WAIT”, the target issues
    - `shmem_putmem(plan->offset[me], ..., tgt)`;
    - `shmem_fence();`
    - `shmem_int_p(plan->remote_status[me], READY, tgt);`
  - Unpack starts when status is read as READY in a while loop over all statuses
Conversion: signaling exposure

- The same remap plan is reused multiple times
  - Risk of scratch buffer overwrite between different axis
- Multiple possible strategies
  - Bulk synchronization (Barrier)
  - Fine grain synchronization (per-scratch readiness with shmem_int_p)
Conversion: signaling exposure

- The same remap plan is reused multiple times
  - Risk of scratch buffer overwrite between different axis
- Multiple possible strategies
  - Bulk synchronization (Barrier)
  - Fine grain synchronization (per-scratch readiness with shmem_int_p)

Technical shortcoming: Shmem_barrier_all takes a fixed stride PE distribution

The group of PE participating in a plan are not distributed that way, we have to use MPI_Barrier to cover the right group....
Conversion: signaling exposure

- The same remap plan is reused multiple times
  - Risk of scratch buffer overwrite between different axis
- Multiple possible strategies
  - Bulk synchronization (Barrier)
  - Fine grain synchronization (per-scratch readyness with shmem_int_p)

This sync pattern looks like 2 sided, but:
1. Rendez-vous to notify exposure comes earlier
2. Opportunistic unpacking can overlap “target not ready” wait time with unpack computation
Taking advantage of injection rate

- Remove packing
  - multiple small puts
  - 1 less memory copy
  - Bulk put-completion synchronization with one fence();
    shmem_int_p();

PE0 - Sender

PE1 - Receiver

FFT_x

Shmem_put
Shmem_put
Shmem_put
Shmem_fence
Shmem_int_p

Data_p1
Data_p2
Data_p3

Signal

Scratch
unPack

out

MPI_Barrier()

MPI_Barrier()

FFT_y

Everyone’s Scratch is Ready

NoPack
Performance

• Titan Cray XK7 supercomputer (@ORNL)
  • Cray MPICH 6.3.0, Cray-shmem 6.3.0
  • Rhodopsin protein input problem, 8x8x8x32k atoms (strong scaling) or 32k atoms per PE (weak scaling)
Pack/NoPack

(a) Strong Scaling. (b) Weak Scaling.

Fig. 3: Total LAMMPS execution time comparison between the following versions: original MPI, HybridBarrier, and NoPack (with and w/o non-blocking shmem_put).

10% impr overall, only 30% communication time: speedup is huge!

Std. dev of results similar

w/o non-blocking put, nopack is slow
Fig. 4: Total LAMMPS execution time comparison between the following versions: original MPI, ChkBuff, and AdvChkBuff.

(a) Strong Scaling.

(b) Weak Scaling.

Fine grain signaling SLOWER than barrier sync.

Overlap between chkbuff and unpack can delay puts.
Conclusions

- Accelerated the `remap3d` routine in LAMMPS with OpenSHMEM

  - Shmem does improve performance vs MPI
    - No costly handling of unexpected messages, no internal buffering, early exposure of “recv” buffers, bulk synchronization, etc.

- Missing features (but soon?)
  - Weird structure for groups on which collective operate, porting MPI code with collective on MPI groups is hard!
  - `shmem_put` too synchronizing by default: if one can relax put-completion semantic at the origin, huge performance gains
  - Missing `shmem_put_notify` (a proposal is ongoing in the std body)

- Hard to predict performance
  - Explicit handling of all synchronization can be cumbersome to end-users
  - Default bulk synchronization not available (see missing features) => one has to implement it himself
  - Bad handling of synchronization can get worse performance than implicit sync. in MPI
  - Often hard to predict the best strategy, performance portability diminished