Achieving Computation-Communication Overlap with Overdecomposition on GPU Systems

To appear at ESPM2 workshop at SC’20

Jaemin Choi

PhD Candidate in Computer Science
University of Illinois Urbana-Champaign

Oct 21, 2020
Overview

• Increasing gap between single-node computational power and inter-node communication performance on modern supercomputers

• Can be tackled from at least 2 directions
  1. Improve communication performance itself with software optimizations and better utilization of hardware support (e.g. GPUDirect, SHARP, hardware tag-matching)
  2. Reduce impact of communication on overall performance (e.g. computation-communication overlap)

• Focus on computation-communication overlap
Overdecomposition

Per-process decomposition (MPI)

4 CPU cores

Overdecomposition (Charm++)
Asynchronous Message-Driven Execution

Chares

PE

Message queue
GPU Execution in Charm++

1. Asynchronously offload work to GPU

2.1. Scheduler progresses communication and executes next chare
2.2. GPU work completes, runtime enqueues new message

3. Work that depends on the completed GPU work can continue (e.g. another entry method of the original chare)
Support asynchronous progress in the runtime

- Avoid synchronization CUDA APIs (e.g. cudaStreamSynchronize)
  - Charm++ scheduler blocked from performing other chares’ work
  - Cannot make forward progress on communication (without comm. threads)
- Directly using CUDA async APIs to determine completion is infeasible
  - Scheduler-driven execution in Charm++
  - CUDA-generated thread disassociated from the Charm++ runtime

   `hapiAddCallback(cudaStream_t stream, CkCallback* callback)`
  - Allows user to schedule a Charm++ callback to be invoked when GPU operations complete in the specified CUDA stream
  - Two compile-time configurable mechanisms based on CUDA Callback and CUDA Events (default)

2. **Prioritize communication-related GPU operations in the application**
   - Single CUDA stream per chare: *delays in communication-related operations* (host-device data transfers, packing/unpacking kernels) due to computational kernels offloaded from other chares to the same GPU
   - Need *separate streams for compute and communication* (with *higher priority* for communication)
   - More complex design may be necessary, as for MiniMD (described in paper)
Achieving Computation-Communication Overlap

(a) Single CUDA stream per char. Communication is delayed by a computational kernel enqueued from another char, causing idle time between iterations.

(b) Separate compute/communication CUDA streams per char, with the communication stream given higher priority. Iterations continue without idle times in between.

Fig. 3. Execution timelines of Jacobi2D with four chars mapped to a single GPU.
Evaluation Platforms

- OLCF Summit
  - 6 NVIDIA Tesla V100s per node
- LLNL Lassen
  - 4 NVIDIA Tesla V100s per node
- PAMILRTS, SMP version of Charm++
- 1 process with 1 PE/core per GPU
  - e.g. 6 PEs and 6 GPUs per compute node on Summit
Benchmarks

• Iterative proxy apps

• Jacobi3D
  • Jacobi iteration performed on 3D grid, overdecomposed into chares
  • Near-neighbor exchange of halo data (up to 6 neighbors)

• MiniMD
  • Proxy app for LAMMPS molecular dynamics code
  • Converted MPI-Kokkos to Charm++-Kokkos
  • CUDA-aware MPI converted to explicit host-device transfers and host messages
  • Kokkos responsible for computational kernels and intra-process data movement
  • Neighbor exchange of atoms, Lennard-Jones force calculation
Performance Results – Jacobi3D

Fig. 5. Performance of Jacobi3D with varying overdecomposition factors on a single node of OLCF Summit.
Performance Results – Jacobi3D

(a) Weak scaling on Summit.
(b) Weak scaling on Lassen.
(c) Strong scaling on Summit.
(d) Strong scaling on Lassen.

Fig. 6. Weak & strong scaling performance of Jacobi3D.
Performance Results – MiniMD

Fig. 7. Weak & strong scaling performance of MiniMD.
Conclusion

• Up to **50%** and **47%** improvement in overall performance with Jacobi3D and MiniMD, respectively.

• With careful design of the application to prioritize communication and support for asynchronous progress of GPU work in the runtime system, computation-communication overlap can significantly improve performance (esp. in weak scaling).

• Future work: improve communication performance with GPU messaging.
GPU Messaging in Charm++ 6.11

• Direct data transfer between GPUs using GPUDirect & CUDA IPC, bypassing host memory

• Currently supports *intra-node* messages, support for inter-node coming soon

• Regular API
  • For point-to-point messages between chares
  • Currently undergoing performance optimizations
  • Included in 6.11-beta as experimental feature
  • Similar to Zerocopy Post Entry Method API, sender sends metadata & receiver performs a get

• Persistent API
  • For persistent P2P messages between chares (reuse of GPU buffers)
  • Useful for iterative applications
  • Will also be part of 6.11 (merged for 2nd beta)
GPU Messaging Performance

- **Charm-H**: Host-staged / **Charm-D**: Regular GPU messaging / **Charm-P**: Persistent GPU messaging
- **OSU latency benchmark (CUDA-aware MPI and Charm++ versions)** on OLCF Summit
Thank you! Questions?