Implementation of Programming Models / Environments

Anirudh Jonnadula
Papers

Exploiting and Evaluating OpenSHMEM on KNL Architecture
(J. Hashmi, M. Li, H. Subramoni, and D. K. Panda, OpenSHMEM workshop, Aug 2017)

Unifying UPC and MPI runtimes: experience with MVAPICH

Designing Scalable Graph500 Benchmark with Hybrid MPI+OpenSHMEM Programming Models
(J. Jose, S. Potluri, K. Tomko and D. K. Panda, ISC, 2013)
Exploiting and Evaluating OpenSHMEM on KNL Architecture
Partitioned Global Address Space (PGAS) Models

- **Key abstraction**
  - Shared memory abstraction over distributed system images

- **Library-level solutions**
  - OpenSHMEM
  - Global Arrays
  - UPC
  - ...
Intel Knights Landing Processor – Overview

• Multi-threaded cores
  – Up to 72 cores (model 7290)
• All cores divided into 36 Tiles
• Each tile contains two core
  – 2 VPU per core
  – 1MB shared L2 cache
• 512-bit wide vector registers
  – AVX-512 extensions
Intel Knights Landing Processor – Overview

- 4 threads per core with Simultaneous Multi-threading
  - Shared and dynamically partitioned resources
  - Out of order execution

- Configurable mesh interconnect
  - AlltoAll: addresses are uniformly distributed across tag directories
  - Quadrant: memory appears as single NUMA domain
  - SNC: memory appears as distinct NUMA domains
• On-package Multi Channel DRAM (MCDRAM)
  - 450 GB/s of theoretical bandwidth (4x of DDR)
  - Configurable in Flat, Cache, and Hybrid modes
Motivation

- Optimizing HPC runtimes and applications on emerging manycores is of great research interest
- Exploring benefits of the architectural features of KNL for OpenSHMEM model and application
  - Impact of vectorization on application kernels
  - MCDRAM vs. DDR performance
  - Exploiting hardware multi-threading
Problem Statements

Can PGAS models, specifically OpenSHMEM, benefit from the architectural features of KNL processor at micro-benchmarks as well as application level?

Can we identify the optimizations that could help achieve better performance on KNL?
Contributions

- Evaluate OpenSHMEM point-to-point, collective, and atomic operations on KNL architecture
- Evaluate OpenSHMEM application kernels
  - Discuss the impact of MCDRAM and vectorization
  - Comparison of KNL against Broadwell on core-by-core and node-by-node performance
- Discuss potential optimization for KNL architecture
Benchmarks and Configurations

- OpenSHMEM Microbenchmark Evaluation
  - Point-to-point (Put/Get), collectives (broadcast, reduce), and atomics
  - OSU Microbenchmark v5.3.2

- OpenSHMEM NAS Parallel Benchmark Kernels
  - MG, BT, EP, SP

- University of Houston OpenSHMEM test suite
  - Five application Kernels
    - 2D-Heat, Heat-Image, Matrix Multiplication, DAXPY, and ISx

- All experiments evaluate KNL 7250 against Broadwell
- Application evaluation on KNL also discuss MCDRAM and AVX-512 benefits
Microbenchmark Evaluations (Intra-node Put/Get)

- Broadwell shows about 3X better performance than KNL on large message
- Muti-threaded memcpy routines on KNL could offset the degradation caused by the slower core on basic Put/Get operations
Inter-node small message latency is only 2X worse on KNL. While large message performance is almost similar on both KNL and Broadwell.
Microbenchmark Evaluations (Collectives)

- Shmem_reduce on 128 processes
  - 2 KNL nodes (64 ppm) and 8 Broadwell nodes (16 ppm).
  - 4X degradation is observed on KNL using collective benchmarks.
  - Basic point-to-point performance difference is reflected in collectives as well

- Shmem_broadcast on 128 processes
  - ~2X improvement compared to Broadwell
Microbenchmark Evaluations (Atoms)

OpenSHMEM atomics on 128 processes

- Using multiple nodes of KNL, atomic operations showed about 2.5X degradation on compare-swap, and Inc atomics
- Fetch-and-add (32-bit) showed up to 4X degradation on KNL
AVX-512 vectorized execution of BT kernel on KNL showed 30% improvement over default execution while EP kernel didn’t show any improvement.

Broadwell showed 20% improvement over optimized KNL on BT and 4X improvement over all KNL executions on EP kernel (random number generation).
Similar performance trends are observed on BT and MG kernels as well.
On SP kernel, MCDRAM based execution showed up to 20% improvement over default at 16 processes.
On heat diffusion based kernels AVX-512 vectorization showed better performance.
MCDRAM showed significant benefits on Heat-Image kernel for all process counts. Combined with AVX-512 vectorization, it showed up to 4X improved performance.
Application Kernels Evaluation (contd.)

Matrix Multiplication kernel

- Vectorization helps in matrix multiplication and vector operations.
- Due to heavily compute bound nature of these kernels, MCDRAM didn’t show any significant performance improvement.

DAXPY kernel
Application Kernels Evaluation (contd.)

Scalable Integer Sort Kernel (ISx)

- Up to 3X improvement on un-optimized execution is observed on KNL
- Broadwell showed up to 2X better performance for core-by-core comparison
Node-by-node Evaluation using Application Kernels

Application Kernels on a single KNL vs. single Broadwell node

- A single node of KNL is evaluated against a single node of Broadwell using all the available physical cores
- HeatImage and ISx kernels showed better performance than Intel Xeon
Performance Results Summary

(Closer to center is better)
Conclusion

- Comprehensive performance evaluation of MVAPICH2-X based OpenSHMEM over the KNL architecture
  - Intra- and inter-node comparison with Broadwell
  - Microbenchmarks and application kernels
- Observed significant performance gains on application kernels when using AVX-512 vectorization
  - 2.5x performance benefits in terms of execution time
- MCDRAM benefits are not prominent on most of the application kernels
  - Lack of memory bound operations
- KNL showed up to 3X worse performance than Broadwell for core-by-core evaluation
- KNL showed better or on-par performance than Broadwell on Heat-Image and ISx kernels for Node-by-Node evaluation
- The runtime implementations need to take advantage of the concurrency of KNL
  - Multi-threaded OpenSHMEM runtimes
Unifying UPC and MPI Runtimes: Experience with MVAPICH
The Need for a Unified Runtime

- Deadlock when a message is sitting in one runtime, but application calls the other runtime
- Current prescription to avoid this is to barrier in one mode (either UPC or MPI) before entering the other
- Bad performance!!
Problem Statement

• Can we design a communication library for UPC?
  - Scalable on large InfiniBand clusters
  - Provides equal or better performance than existing runtime

• Can this library support both MPI and UPC?
  - Individually, both with great performance
  - Simultaneously, with great performance and less memory
Various Configurations for running UPC and MPI Applications

- Pure UPC or UPC + MPI Applications
  - UPC Compiler
  - GASNet Interface and UPC Runtime
  - GASNet MPI Runtime
  - MVAPICH - MPI Standard Interface
  - MVAPICH - Aptus Runtime
  - InfiniBand Network

- Pure UPC Applications
  - UPC Compiler
  - GASNet Interface and UPC Runtime
  - GASNet IBverbs Runtime
  - GASNet-IBV

- MVAPICH
  - MVAPICH - Aptus Runtime
  - MVAPICH - InfiniBand Network

- GASNet
  - GASNet-MPI
  - GASNet-IBV
  - GASNet-INC
  - GASNet-INC Runtime

Our Contribution
Overall Approach

- Unified runtime provides APIs for MPI and GASNet
- **INCR** ([Integrated Communication Runtime])
Unified Implementation

- All resources are shared between MPI and UPC
  - Connections, buffers, memory registrations
  - Schemes for establishing connections (fixed, on-demand)
  - RDMA for large AMs and for PUT, GET
Experimental Setup

- MVAPICH version 1.1 extended to support INCR
- Berkeley GASNet version 2.10.2 (--enable-pshm)
- Experimental Testbed
  - Type 1
    - Intel Nehalem (dual socket quad core Xeon 5500 2.4GHz)
    - ConnectX QDR InfiniBand
  - Type 2
    - Intel Clovertown (dual socket quad core Xeon 2.33GHz)
    - ConnectX DDR InfiniBand
  - Type 3
    - AMD Barcelona
    - Quad-socket quad-core Opteron 8530 processors
    - ConnectX DDR InfiniBand
Microbenchmark: upc-memput

- Cluster #1 used for these experiments
- GASNet-INCR performs identically with GASNet-IBV
- Comparatively GASNet-MPI performs much worse
- Mismatch of Active Message semantics
  - Message queue processing overheads
Microbenchmark: upc_memget

- GASNet-INCR performs identically with GASNet-IBV
- Due to mismatch of AM semantics with MPI leads to worse performance
Memory Scalability

- UPC “hello world” program
- GASNet-IBV establishes all-to-all reliable connections
  - Not scalable (may be improved in future release)
- GASNet-INCR best scalability due to inherent Aptus design
- Cluster #2 used for this experiment
Evaluation using UPC NAS Benchmarks

- GASNet-INCR performs equal or better than GASNet-IBV
- 10% improvement for CG (B, 128)
- 23% improvement for MG (B, 128)
- Cluster #3 used for these experiments
Evaluation using Hybrid NAS-FT

- Modified NAS FT UPC all-to-all pattern using MPI_Alltoall
- Truly hybrid program
- 34% improvement for FT (C. 128)
- Cluster #3 used for this experiment
Conclusions and Future Work

- Integrated Communication Runtime (INCR): supports MPI and UPC simultaneously
- Promising: MPI communication not harmed and UPC communication not penalized
- No need for programmer to barrier between UPC and MPI modes, as is current practice
- Pure UPC NAS: 10% improvement CG (B, 128), 23% improvement MG (B, 128)
- MPI+UPC FT: 34% improvement for FT (C, 128)
- Public release with MVAPICH2 coming soon
Designing Scalable Graph500 benchmark with Hybrid MPI + OPENSHMEM Programming Models
Hybrid (MPI+PGAS) Programming for Exascale Systems

- Application sub-kernels can be re-written in MPI/PGAS based on communication characteristics
- Benefits:
  - Best of Distributed Computing Model
  - Best of Shared Memory Computing Model
- Exascale Roadmap*:
  - “Hybrid Programming is a practical way to program exascale systems”

Introduction to Graph500

- Graph500 Benchmark
  - Represents data intensive and irregular applications that use graph algorithm-based processing methods
  - Bioinformatics and life sciences, social networking, data mining, and security/intelligence rely on graph algorithmic methods
  - Exhibits highly irregular and dynamic communication pattern
  - Earlier research have indicated scalability limitations of the MPI-based Graph500 implementations
Problem Statement

- Can a high performance and scalable Graph500 benchmark be designed using MPI and PGAS models?
- How much performance gain can we expect?
- What will be the strong and weak scalability characteristics of such a design?
Graph500 Benchmark – The Algorithm

• Breadth First Search (BFS) Traversal
• Uses ‘Level Synchronized BFS Traversal Algorithm
  – Each process maintains – ‘CurrQueue’ and ‘NewQueue’
  – Vertices in CurrQueue are traversed and newly discovered vertices are sent to their owner processes
  – Owner process receives edge information
    • if not visited; updates parent information and adds to NewQueue
  – Queues are swapped at end of each level
  – Initially the ‘root’ vertex is added to currQueue
  – Terminates when queues are empty
• Size of graph represented by SCALE and Edge Factor (EF)
  – #Vertices = 2**SCALE, #Edges = #Vertices * EF
MPI-based Graph500 Benchmark

- MPI_Isend/MPI_Test-MPI_Irecv for transferring vertices
- Implicit barrier using zero length message
- MPI-AllReduce to count number newqueue elements
- Major Bottlenecks:
  - Overhead in send-receive communication model
    - More CPU cycles consumed, despite using non-blocking operations
    - Most of the time spent in MPI-Test
  - Implicit Linear Barrier
    - Linear barrier causes significant overheads
- Other MPI Implementations
  - MPI-CSR, MPI-CSC, MPI-OneSided
Design Challenges for Hybrid Graph500

• Co-ordination between sender and receiver processes and between multiple sender processes
  – How to synchronize, while using one-sided communication?
• Memory scalability
  – Size of receive buffer
• Synchronization at the end of each level
  – Barrier operations simply limit computation-communication overlap
• Load imbalance
Detailed Design

- Communication and co-ordination using one-sided routines and fetch-add atomic operations
- Buffer structure for efficient computation-communication overlap
- Level synchronization using non-blocking barrier
- Load Balancing
Communication and Co-ordination

- Vertices transferred using OpenSHMEM shmem_put routine
- Receive buffers are globally shared
- Receive buffer size depends on number of local edges that the process owns and connectivity
  - Size is independent of system scale
- Atomic fetch-add operation for co-ordinating between sender and receiver, and between multiple senders
  - Receive buffer indices are globally shared
Buffer Structure for better Overlap

- Receiver process shall know if the data has arrived
- Buffer structure helps to identify incoming data
- Receive process ensures arrival of complete data
- Packet by checking tail marker and can then process immediately
Level synchronization using non-blocking barrier

- MPI-3 non-blocking barrier for level synchronization
- Process enters the barrier and still can continue to receive and process incoming vertices
- Offers better computation/communication overlap
Intra-node Load Balancing

- Overloaded process exposes work
- Idle process takes up shared work and processes it, and puts back for post-processing
- Uses ‘shmemp_ptr’ routine in OpenSHMEM to access shared memory data
Experiment Setup

• Cluster A (TACC Stampede)
  – Intel Sandybridge series of processors using Xeon dual 8 core sockets (2.70GHz) with 32GB RAM
  – Each node is equipped with FDR ConnectX HCAs (54 Gbps data rate) with PCI-Ex Gen3 interfaces

• Cluster B
  – Xeon Dual quad-core processor (2.67GHz) with 12GB RAM
  – Each node is equipped with QDR ConnectX HCAs (32Gbps data rate) with PCI-Ex Gen2 interfaces

• Software Stacks
  – Graph500 v2.1.4
  – MVAPICH2-X OpenSHMEM (v1.9a2) and OpenSHMEM over GASNet (v1.20.0) and
Graph500 - BFS Traversal Time

- Hybrid design performs better than MPI implementations
- 4,096 processes
  - 2.2X improvement over MPI-CSR
  - 5X improvement over MPI-Simple
- 8,192 processes
  - 7.6X improvement over MPI-Simple (Same communication characteristics)
  - 2.4X improvement over MPI-CSR
Unified Runtime vs. Separate Runtimes

- Hybrid-GASNet uses separate runtimes for MPI and OpenSHMEM
  - Significant performance degradation due to lack of efficient atomic operations, and overhead due to separate runtimes
- For 1,024 processes
  - BFS time for Hybrid-GASNet: 22.8 sec
  - BFS time for Hybrid MV2X: 0.58 sec
Load Balancing

- Evaluations using HPC Toolkit indicate that load is being balanced within node
- Load balancing limited within a node
  - Need for post processing
  - Higher cost for moving data
- Amount of work almost equal at each rank!
Scalability Analysis

- **Strong Scaling**
  Graph500 Problem Scale = 29

- **Weak Scaling**
  Graph500 Problem Scale = 26 per 1,024 processes

- Results indicate good scalability characteristics
Conclusion & Future Work

- Presented a scalable design of Graph500 benchmark using hybrid MPI+OpenSHMEM
- Identified critical bottlenecks in the MPI-based implementation
- Not intended to compare programming models, but demonstrate the benefits of hybrid model
- Performance Highlights
  - At 16,384 cores, Hybrid design achieves $13 \times$ improvement over MPI-Simple and $2.4 \times$ improvement over MPI-CSR
  - Exhibits good scalability characteristics
  - Significant performance improvement over using separate runtimes
- Plan to improve load-balancing scheme, considering inter-node
- Plan to evaluate our design at larger scales and also consider real-world applications