CHAPEL 1.31/1.32 RELEASE NOTES: SS-11 / IB PERFORMANCE STATUS

Chapel Team
June 22, 2023 / September 28, 2023
OUTLINE

• Recent Scalability Highlights
• Performance Background
• Aries Performance
• Slingshot-11 Performance
• InfiniBand Performance
• Slingshot vs InfiniBand
• Next Steps
RECENT SCALABILITY HIGHLIGHTS
We had brief access to larger machines to evaluate performance at higher scales

- In May 2021 we had access to a 576 node HDR-100 InfiniBand machine
  - Collected Arkouda argsort results
- In May 2023 we had access to an 8,192 node Slingshot-11 machine
  - Collected Arkouda argsort and Bale indexgather results

All Chapel results are with 1 process per node using a single NIC
ARKOUDA SCALABILITY HIGHLIGHTS

HPE Apollo (May 2021)
- HDR-100 Infiniband network (100 Gb/s)
- 576 compute nodes
- 72 TiB of 8-byte values
- ~480 GiB/s (~150 seconds)

HPE Cray EX (April 2023)
- Sling114,688 network (200 Gb/s)
- 896 nodes
- 28 TiB of 8-byte values
- ~1200 GiB/s (~24 seconds)

HPE Cray EX (May 2023)
- Slingshot-11 network (200 Gb/s)
- 8192 nodes
- 256 TiB of 8-byte values
- ~8500 GiB/s (~31 seconds)

A notable performance achievement in ~100 lines of Chapel
ARKOUDA SCALABILITY HIGHLIGHTS

HPE Apollo (May 2021)
- HDR-100 Infiniband network (100 Gb/s)
- 576 compute nodes
- 72 TiB of 8-byte values
- ~480 GiB/s (~150 seconds)

HPE Cray EX (April 2023)
- Slingshot-11 network (200 Gb/s)
- 896 compute nodes
- 28 TiB of 8-byte values
- ~1200 GiB/s (~24 seconds)

A notable performance achievement in ~100 lines of Chapel

Arkouda Argsort Performance

Slingshot-11 April 2023, 32 GiB/node
HDR-100 IB May 2021, 128 GiB/node

better

A notable performance achievement in ~100 lines of Chapel
ARKOUDA SCALABILITY HIGHLIGHTS

HPE Apollo (May 2021)
- HDR-100 Infiniband network (100 Gb/s)
- 576 compute nodes
- 72 TiB of 8-byte values
- ~480 GiB/s (~150 seconds)

HPE Cray EX (April 2023)
- Slingshot-11 network (200 Gb/s)
- 896 compute nodes
- 28 TiB of 8-byte values
- ~1200 GiB/s (~24 seconds)

HPE Cray EX (May 2023)
- Slingshot-11 network (200 Gb/s)
- 8192 compute nodes
- 256 TiB of 8-byte values
- ~8500 GiB/s (~31 seconds)

A notable performance achievement in ~100 lines of Chapel
INDEXGATHER SCALABILITY HIGHLIGHTS

- Exstack performance is ahead up to 64 nodes, but Chapel is ahead above that
  - Exstack library and Conveyors implementation of indexgather only support a fixed number of PEs
  - Conveyors can be adjusted to support more, but only determined that later

![Bale Indexgather Performance](image)

![Bale Indexgather Performance](image)
INDEXGATHER SCALABILITY HIGHLIGHTS

- Chapel indexgather runs and scales up to 4K nodes
  - Timed out gathering results, but uses same aggregators as Arkouda so expected to scale to 8K nodes as well
SCALABILITY HIGHLIGHTS SUMMARY

• We are happy with these scalability results, but they are Arkouda/aggregation focused
  • Aggregation only covers a relatively narrow set of network operations
• We wanted to take a step back and evaluate overall performance using our core benchmarks
  • These cover a broader range of HPC idioms
PERFORMANCE BACKGROUND
**PERFORMANCE BACKGROUND**

Platforms

- We want benchmark results to evaluate Chapel’s performance on InfiniBand and Slingshot networks
  - As well as some reference MPI/SHMEM figures to get a sense of what the hardware is capable of

- We have only been optimizing for InfiniBand and Slingshot-11 since 2020
  - Prior to that, we focused on performance for Cray XC systems with the Aries network
    - Intent was to ensure Chapel had the right language features/semantics first, then optimize for other networks

- Hardware has changed dramatically between XC systems and modern InfiniBand/Slingshot systems
  - Typical XC was dual socket with ~36 intel cores per node and a single NIC per node
    - Penalty for getting NUMA affinity wrong was relatively low (1.5-2x)
  - Modern systems we’ve run on are dual socket with 128+ AMD cores or 72+ Intel cores, often with multiple NICs
    - Penalty for getting NUMA affinity wrong can be quite high (~10x)
    - Hard to target multiple NICs from a single process/locale
Co-locales

- Historically, Chapel ran with a process/locale per node, multithreading within a node
  - This worked well on single-NIC systems with lower cross-socket NUMA penalties
  - Especially when we could provide first-touch for large arrays on Aries systems

- Modern hardware performs best with a process per socket or even NUMA domain
  - Due to high cost of getting NUMA affinity wrong and benefit of targeting multiple NICS from different processes

- Much of our recent work has been to support running Chapel with multiple processes/locales per node
  - We refer to this as co-locales: co-located / cooperative locales running on the same node
  - Initial work has focused on a process per socket, but have foundations to support more arbitrary mappings

- Results will include data for 1 and 2 locales per node (LPN)
PERFORMANCE BACKGROUND
System Configurations

• We wanted to collect core benchmark results on systems with similar per-node hardware
  • Have easy access to single-NIC systems, so no multi-NIC results, but allows easier comparison between IB/SS

• Slingshot-11/InfiniBand hardware
  • Dual-socket Milan (128-cores total)
  • Single 200 Gbps NIC (HDR-200 and Slingshot-11)

• Historical Cray XC hardware
  • Dual-socket Intel (36-cores total)
  • Single Aries NIC
**PERFORMANCE BACKGROUND**

Benchmark Description

- **Core Benchmarks**
  - **Stream**: No communication, NUMA affinity sensitive
  - **ISx**: Concurrent bulk communication over wide address range, NUMA affinity sensitive
  - **RA**: Concurrent random fine-grained communication over wide address range
  - **Indexgather**: Concurrent bulk communication between small address range
- **Arkouda argsort**
  - Concurrent bulk communication between small address range, some NUMA affinity sensitivity

- Unless otherwise noted, results show weak scaling (fixed problem size per node)
PERFORMANCE BACKGROUND

Reference Background

• Collecting reference numbers on HPE/Cray systems is straightforward
  • cray-mpich and cray-openshmemx/cray-shmem are tuned and readily available
  • SHMEM expects multiple NICs, might not be fully tuned for single
    – Even so, still a useful comparison point to see if Chapel performance is way off

• We hit challenges collecting InfiniBand reference numbers that we could feel confident in
  • No vendor-supported MPI/SHMEM implementation on the particular system we ran on
    – Tried MPICH, OpenMPI and MVAPICH2 for MPI; MVAPICH2 had best performance but still seemed low
    – OpenMPI was the only available SHMEM implementation, but performance was quite poor
  • Opted to not include reference numbers for InfiniBand at this time
ARIES PERFORMANCE
We use ‘CHPL_COMM=ugni’ to target Aries

- Dynamically allocates large arrays from the OS and registers them at allocation time
  - Enables first-touch for each array, provides good NUMA affinity and limits fragmentation
  - Communication is fast since all memory is registered, able to achieve hardware bandwidth injection rates

- Each runtime thread has a communication endpoint
  - Enables concurrent and uncontended injection, achieves hardware injection rates for small messages

- Has support for network atomics

- Uses a blocking active-message handler, avoids interference when there are no incoming AMs

In general, our core benchmarks were competitive with or ahead of reference MPI/SHMEM codes

- Core benchmarks tested up to 256 nodes
- Previous scaling studies done up to 2048 nodes
ARIES PERFORMANCE

Performance Results

STREAM Performance (GB/s)

RA Performance (GUPS)

ISx Time (seconds)

Bale Indexgather Performance
We use ‘CHPL_COMM=ofi’ with the cxi provider to target Slingshot-11

- Uses a fixed and statically registered heap
  - Memory is interleaved across the NUMA domains that a locale is running on
  - Static registration makes RDMA trivial (all memory registered at program startup)
- Supports running a process per socket
  - Limits interleaving to a single socket
  - Does not currently support shared memory bypass for co-located locales
- Each runtime thread has a communication endpoint
  - Enables concurrent and hopefully uncontended injection (fast concurrent fine-grained communication)
- Has support for network atomics
- Uses a polling active-message handler, can cause interference even when there are no incoming AMs
SLINGSHOT-11 PERFORMANCE
Stream Results

- 1-locale-per-node performance is limited by the fixed heap being interleaved across sockets
- Expected locale-per-socket to achieve near peak performance (like we saw with gasnet-ibv on IceLake)
  - However, additional NUMA behavior within AMD CPUs seems to limit performance
    - Additionally, seeing even worse performance when using transparent huge pages (THP)

Stream Performance
192 GiB/node

GB/s

MPI+OpenMP
Chapel 2 LPN (no THP)
Chapel 2 LPN
Chapel 1 LPN

Nodes (128 cores / node)

better
SLINGSHOT-11 PERFORMANCE
Stream Next Steps

• Investigate performance differences from transparent huge pages
• Add support for dynamic heap/registration, enabling first-touch affinity
• Add support for running with a locale per NUMA domain
  • Results in ideal NUMA affinity without requiring first-touch
  • But increases number of AM handlers and number of peers
ISx Results

- ISx performance is fairly competitive, slightly behind under 32 nodes
- Believe communication patterns are optimized, likely cause is imperfect NUMA affinity
Fine-grained Indexgather Results

- Fine-grained indexgather performance is behind SHMEM
  - Root cause unknown, need to investigate
SLINGSHOT-11 PERFORMANCE
Aggregated Indexgather Results

- Aggregated indexgather performance is generally ahead of SHMEM
  - Chapel implementation is more asynchronous, and task-based runtime enables trivial comm/compute overlap
  - Additionally, Chapel using a process per node/socket reduces number of peers, enabling fewer and larger buffers
SLINGSHOT-11 PERFORMANCE
Thread Scaling Results

- GETs have good thread scaling, but PUTs have poor scaling when using FI_INJECT
  - Likely caused by some resource exhaustion; debug messages show running out of resources for FI_INJECT
    - Running without inject we see better scaling, but low per-core rate
  - Have some leads, but root cause is not fully understood
INFINIBAND PERFORMANCE
Implementation Background

- We use ‘CHPL_COMM=gasnet’ with the ibv conduit to target InfiniBand
  - Two production mechanisms for registering memory
    - Segment fast (like SS-11): fixed heap, all memory is registered at program startup
      Memory is interleaved
      Static registration enables fast communication
    - Segment large (default): fixed heap, but memory is registered at communication time in 128K chunks
      Global first-touch (fixed heap results in memory reuse)
      Dynamic registration in small chunks can limit communication performance
- Supports running a process per socket
  - We currently disable GASNet shared memory bypass for co-located locales (would prevent blocking AM handler)
- Runtime threads share a single communication endpoint
  - Serializes communication injection, hurts fine-grained communication
- Not currently utilizing the GASNet-EX remote atomic API
  - And GASNet does not currently offload atomics on InfiniBand anyway
- Uses a blocking active-message handler, avoids interference when there are no incoming AMs
Dynamic registration can achieve peak stream performance:
- Static registration suffers from the same NUMA penalties as SS-11 (including THP issues)
INFINIBAND PERFORMANCE
ISx Results

- ISx performance suffers with dynamic registration
  - Registration at comm time and 128K chunking limits bandwidth over a wide address range
- Performance with static registration is much better
  - Locale per socket improves affinity and increases communication concurrency
**INFINIBAND PERFORMANCE**

Indexgather Results

- Indexgather is largely unimpacted by registration type
  - Lots of communication, but between same addresses so dynamic registration overhead is amortized

![Bale Indexgather Performance](image)

- Static Register 2 LPN
- Static Register 1 LPN
- Dynamic Register 2 LPN
- Dynamic Register 1 LPN

**Bale Indexgather Performance**

32 GiB/node, $2^{32}$ requests/node
INFINIBAND PERFORMANCE
Thread Scaling Results

- GETs and PUTs have poor thread scaling
  - Due to injection serialization
SLINGSHOT VS INFINIBAND
SLINGSHOT VS INFINIBAND
Stream Results

- GASNet dynamic registration matches reference
  - Static registration suffers from suboptimal NUMA affinity
ISx Results

- ISx performance is similar on InfiniBand and Slingshot-11 up to 16 nodes
  - InfiniBand falling off above that, need to investigate root cause
    - Reminder that InfiniBand dynamic registration (not shown) had substantially worse performance

**SLINGSHOT VS INFINIBAND**

![Graph showing ISx Time vs Nodes (128 cores / node)]

- chapel IB Stat 2 LPN
- chapel SS-11 2 LPN
- SHMEM

faster
Indexgather Results

- Indexgather performance is similar on InfiniBand and Slingshot-11
**SLINGSHOT VS INFINIBAND**

Random Access (RMO) Results

- Injection serialization limits fine-grained performance for gasnet-ibv
- Dynamic registration further limits performance for operations over a wide address range
SLINGSHOT VS INFINIBAND
Random Access (atomics) Results

• Injection serialization and lack of offloaded atomics hurts gasnet-ibv performance

![RA-atomics Performance Graph](image-url)
**SLINGSHOT VS INFINIBAND**

Thread Scaling Results

- Serial and up to 4 thread GET rates are similar on InfiniBand and Slingshot-11
- Only higher InfiniBand thread counts start to fall off from serialization
SLINGSHOT VS INFINIBAND
Arkouda Results

- For larger problem sizes, Arkouda performance is similar on InfiniBand and Slingshot-11
Arkouda Strong Scaling Results

- For smaller problem sizes (or strong scaling) gasnet-ibv performance falls off
  - Smaller transfers start seeing overhead of injection serialization
  - Fewer transfers inhibits amortization of dynamic registration cost
**NEXT STEPS**

Benchmarks

- Automate performance scaling runs using new ‘chplExperiment’ framework
  - Publish scripts so users can run on their systems too
- Run additional core benchmarks and gather more reference numbers
  - Collect Chapel and reference results for PRK Stencil and NAS FT
  - Collect reference results for Random Access
- Collect reference numbers on InfiniBand
  - Requires figuring out installs or running on a machine with vendor tuned implementations
- Run core benchmarks at higher scales
- Run core benchmarks on multi-NIC systems
- Run additional user applications like CHAMPS
- Port additional benchmarks like Bale, PRKs
**NEXT STEPS**

Slingshot-11

• Investigate and improve stream performance on transparent huge pages
• Resolve fine-grained performance thread scaling issues
• Add allocation-time dynamic registration and explore ODP
• Evaluate and tune multi-NIC performance
• Expand co-locale support to allow an arbitrary number of locales per node
  • And add support for shared memory bypass
• Explore possibility of using a blocking active message handler
**NEXT STEPS**

**InfiniBand**

- Complete and merge code co-developed with GASNet team to upgrade to GASNet-EX API
  - Minimal transliteration of existing GASNet code—does not make use of new features
- Use GASNet-EX remote atomics (ideally with more offload support from GASNet)
- Improve thread scaling by using GASNet-EX multi-endpoint API (also requires GASNet work)
- Evaluate performance of UCX conduit
- Improve registration story, preferably 1 good default instead of being application-dependent
  - Ideally want either allocation time dynamic registration (with a new GASNet API to register) or ODP
    - But ODP (on demand paging) requires good hardware/firmware support, which we’re not confident about
- Expand co-locale support to allow an arbitrary number of locales per node
  - And enable GASNet’s existing support for shared memory bypass
    - Either requires polling active-message handler or collaboration with GASNet team to only bypass GETs/PUTs
THANK YOU

https://chapel-lang.org
@ChapelLanguage