Job Startup at Exascale: Challenges and Solutions

Overview

Current Trends in HPC
- Tremendous increase in system and job sizes
- Dense many-core systems becoming popular
- Less memory available per process
- Fast and scalable job startup is essential

Importance of Fast Job Startup
- Development and debugging
- Regression/Acceptance testing
- Checkpoint restart

Performance Bottlenecks
Static Connection Setup
- Setting up O(num_procs^2) connections is expensive
- OpenSHMEM, UPC and other PGAS libraries lack on-demand connection management

Network Address Exchange over PMI
- Limited scalability, no potential for overlap
- Not optimized for symmetric exchange

Global Barriers
- Unnecessary synchronization and connection setup

Memory Scalability Issues
- Each node requires O(number of processes * processes per node) memory for storing remote endpoint information

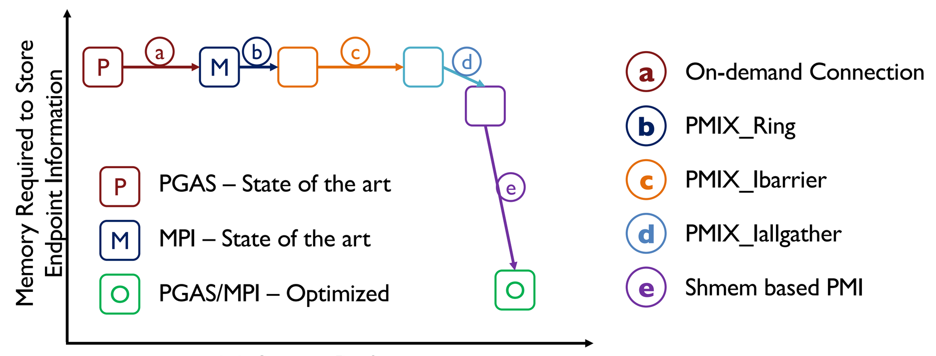
Proposed Solutions
On-demand Connection Management (a)
- Exchange information and establish connection only when two peers are trying to communicate[1]

PMIX_Ring Extension (b)
- Move bulk of the data exchange to high-performance networks[2]

Non-blocking PMI Collectives (c) (d)
- Overlap the PMI exchange with other tasks[3]

Shared-memory based PMI Get/Allgather (e)
- All clients access data directly from the launcher daemon through shared memory regions[4]

Summary
- Near-constant MPI and OpenSHMEM initialization time at any process count
- 10x and 30x improvement in startup time of MPI and OpenSHMEM respectively at 16,384 processes
- Reduce memory consumption by O(ppn)
- 1GB Memory saved @ 1M processes and 16 ppn



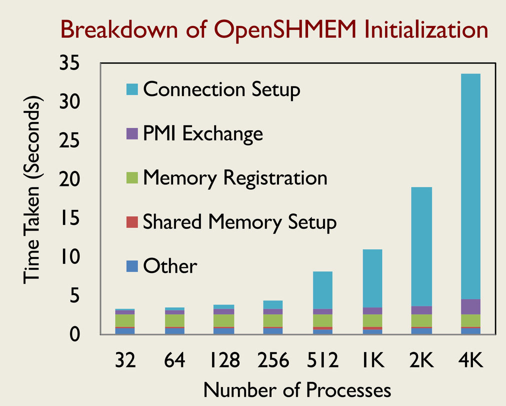
References

[1] On-demand Connection Management for OpenSHMEM and OpenSHMEM+MPI. (Chakraborty et al, HIPS '15)
[2] PMI Extensions for Scalable MPI Startup. (Chakraborty et al, EuroMPI/Asia '14)
[3] Non-blocking PMI Extensions for Fast MPI Startup. (Chakraborty et al, CCGrid '15)
[4] SHMEMPMI – Shared Memory based PMI for Improved Performance and Scalability. (Chakraborty et al, CCGrid '16)

More Information
- Available in latest MVAPICH2 and MVAPICH2-X
- http://mvapich.cse.ohio-state.edu/downloads/
- https://go.osu.edu/mvapich-startup

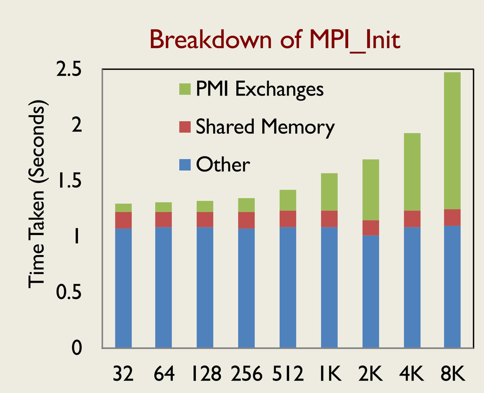
Challenges

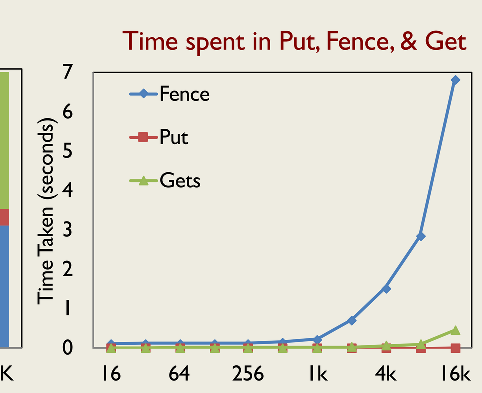
Static Connection Setup
- Setting up connections takes over 85% of the total startup time with 4,096 processes
- RDMA operations require exchanging information about memory segments registered with the HCA



Application	Processes	Average Peers
BT	64	8.7
	1024	10.6
EP	64	3.0
	1024	5.0
MG	64	9.5
	1024	11.9
SP	64	8.8
	1024	10.7
2D Heat	64	5.3
	1024	5.4

Shortcomings of Current PMI design
- Puts and Gets are local operations
- Fence consumes most of the time
- Time taken for Fence grows approximately linearly with amount of data transferred (number of keys)

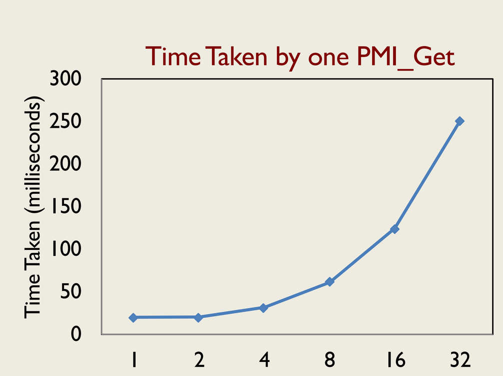


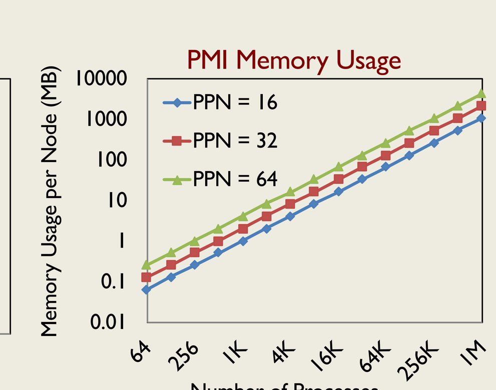


Non-blocking PMI Collectives
- PMI operations are progressed by separate processes handling process management
- MPI library not involved in progressing PMI communication
- Similar to Functional Partitioning approaches
- Can be overlapped with other initialization tasks

PMIX_Request
- Non-blocking collectives return before the operations is completed
- Return an opaque handle to the request object that can be used to check for completion

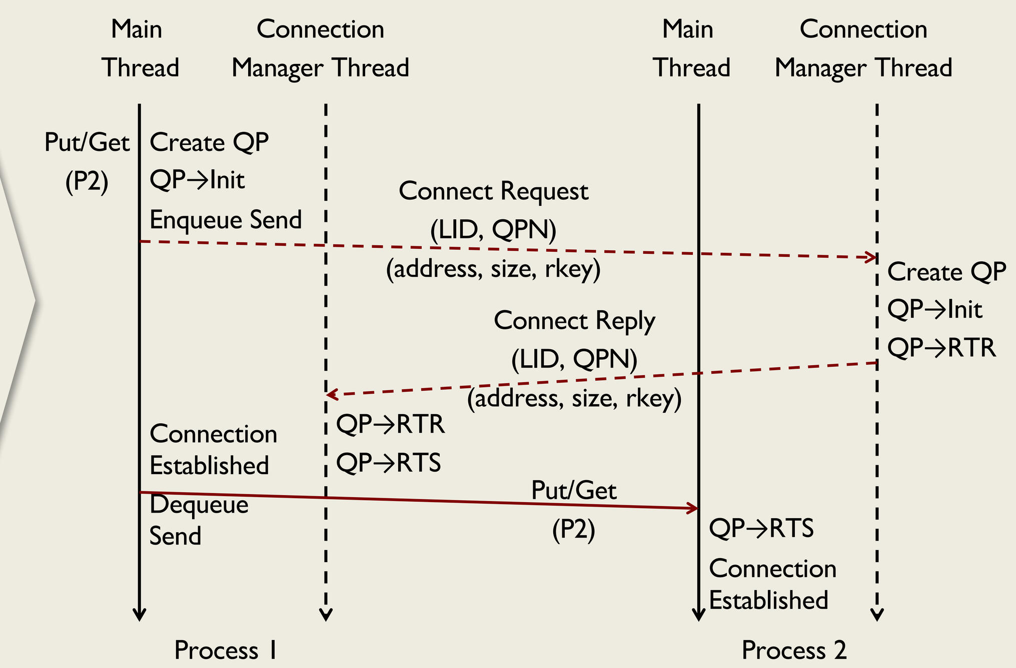
Memory Scalability in PMI
- PMI communication between the server and the clients are based on local sockets
- Latency is high with large number of clients
- Copying data to client's memory causes large memory overhead



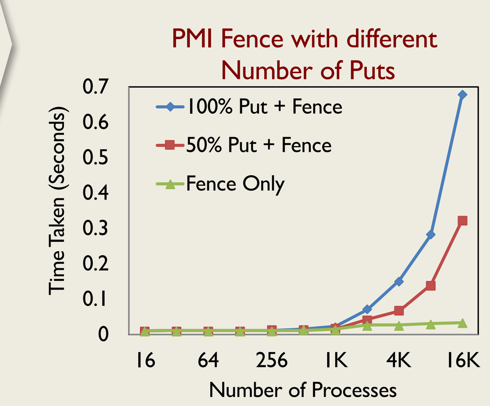


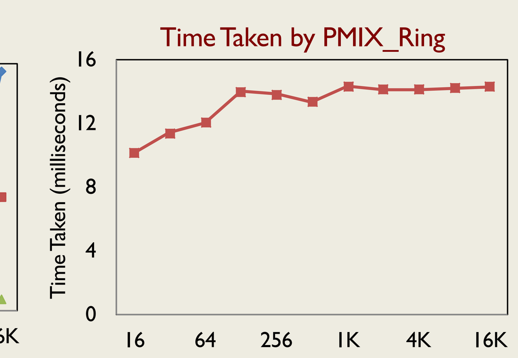
Solution

On-demand Connection Establishment



New Collective – PMIX_Ring
- A ring can be formed by exchanging data with only the left and the right neighbors
- Once the ring is formed, data can be exchanged over the high speed networks like InfiniBand
- int PMIX_Ring(char value[], char left[], char right[], …)



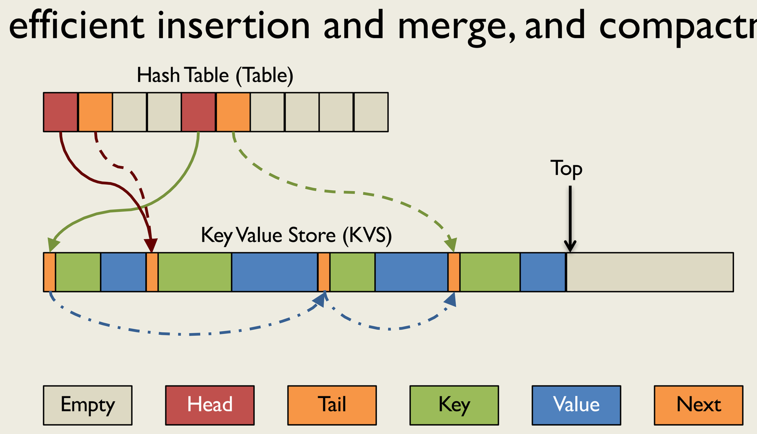


PMIX_KVS_Ifence
- Non-blocking version of PMI2_KVS_Fence
- int PMIX_KVS_Ifence(PMIX_Request *request)

PMIX_Iallgather
- Optimized for symmetric data movement
- Reduces data movement by up to 30%
- 286KB → 208KB with 8,192 processes
- int PMIX_Iallgather(const char value[], char buffer[], PMIX_Request *request)

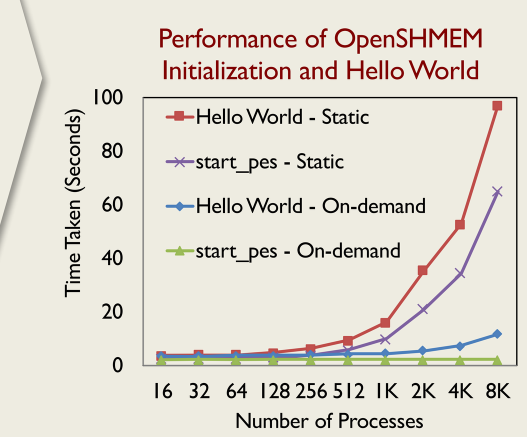
PMIX_Wait
- Wait for the specified request to be completed
- int PMIX_Wait(PMIX_Request request)

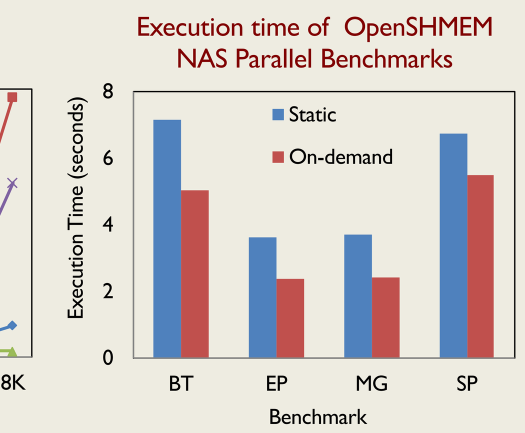
Shared Memory (shmem) based PMI
- Open a shared memory channel between the server and the clients
- A hash table is suitable for Fence while Allgather only requires an array of values
- Use a hash table based on two shmem regions for efficient insertion and merge, and compactness



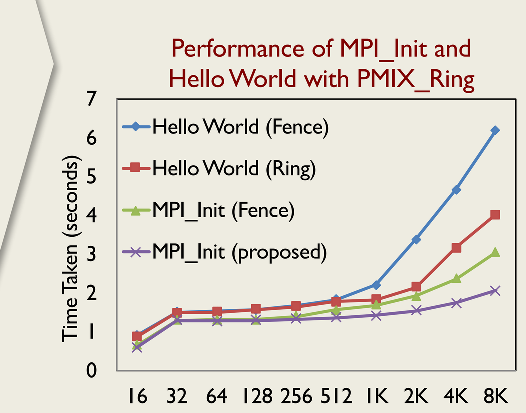
Results

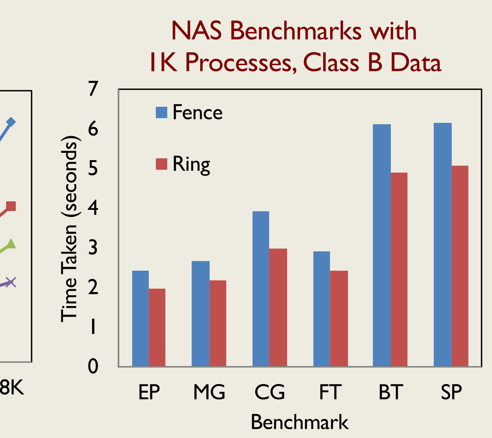
Results – On-demand Connection[1]
- 29.6 times faster initialization time
- Hello world performs 8.31 times better
- Execution time of NAS benchmarks improved by up to 35% with 256 processes and class B data



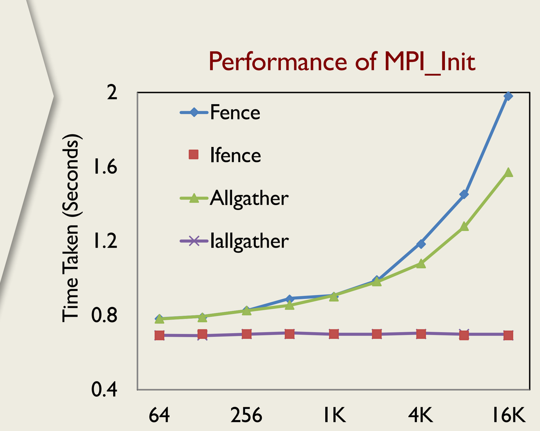


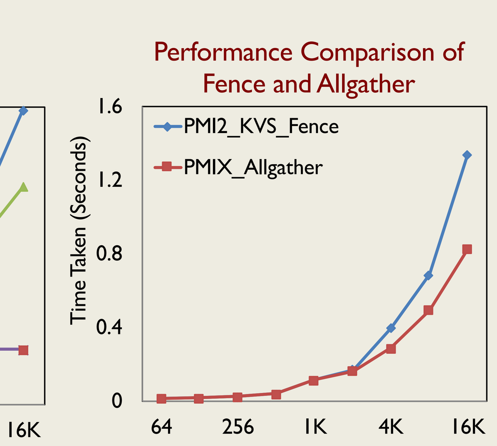
Results – PMI Ring Extension[2]
- MPI_Init based on PMIX_Ring performs 34% better compared to the default PMI2_KVS_Fence
- Hello World runs 33% faster with 8K processes
- Up to 20% improvement in total execution time of NAS parallel benchmarks



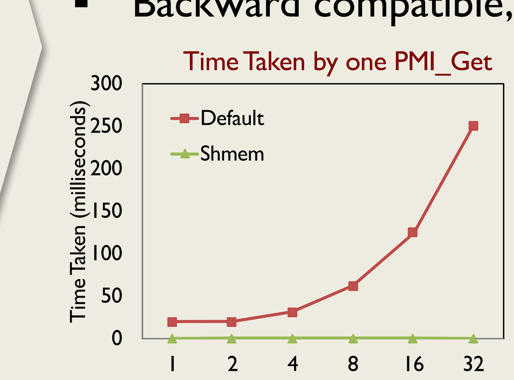


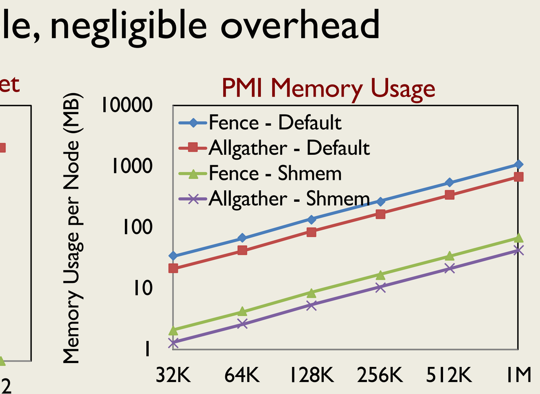
Results - Non-blocking PMI[3]
- Near-constant MPI_Init at any scale
- MPI_Init with Iallgather performs 288% better than the default based on Fence
- Blocking Allgather is 38% faster than blocking Fence





Results – Shared Memory based PMI[4]
- PMI Get takes 0.25 ms with 32 ppn
- 1,000 times reduction in PMI Get latency compared to default socket based protocol
- Memory footprint reduced by O(Processes Per Node) ≈ 1GB @ 1M processes, 16 ppn
- Backward compatible, negligible overhead





Sourav Chakraborty, Dhabaleswar K Panda, (Advisor), The Ohio State University