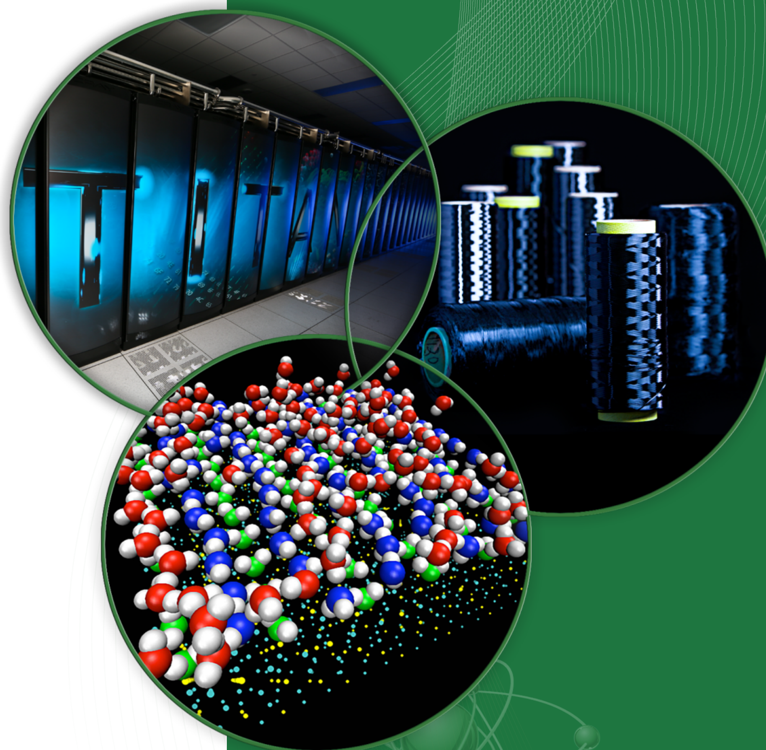
On Synchronisation and Memory Reuse in OpenSHMEM



Aaron Welch, Manjunath Gorentla Venkata

OAK RIDGE
National Laboratory

Teams and *Spaces*

- Proposed *teams* extention promises greater control over asynchronous processing and ability to handle dynamic problems

- Memory available to *teams* remains globally symmetric, cumbersome, and potentially inefficient

- *Spaces* aim to solve the problem by providing an efficient memory solution for *teams* at minimal cost to the application

Examining Use of pSync Arrays

- New extensions
  - Collectives are being enhanced to work with *teams*
  - Memory allocation within *teams* through *spaces*
- Old ways
  - pSync/pWrk arrays still need to be carefully managed by users
- Allocation within *spaces* prevents headache of managing pSync arrays across *teams*
  - Only partly eliminates management burden

Solution: Implementation Manages pSync Arrays

- Focus on pSync (pWrk tuned to specific operations)
- Ability to move pSync management to implementation depends on:
  - Ability to obtain memory without global synchronisation (solved by *spaces*)
  - Ability to determine when memory can be reused (open problem)
- Thus, need to focus on safe memory reuse
  - Three such implementations were designed

pSync Management Goals

- Don't be intrusive
  - No new interfaces
- Low overhead
  - Avoid additional synchronisation/communication costs
  - Minimise memory footprint
- How to do this?
  - There is no golden goose!

How to Reuse pSync: Strategy 1 (Additional Synchronisation)

- Maintain *n* pSync arrays
- Each collective locks and uses a pSync array from the pool of free arrays
- Introduce additional synchronisation to communicate when particular array can be reused
  - Does not need to block
  - When all PEs agree that a pSync is no longer in use, release lock and put the array back in pool
- If free arrays depleted, wait or allocate more

How to Reuse pSync: Strategy 2 (Unlock on User Barrier)

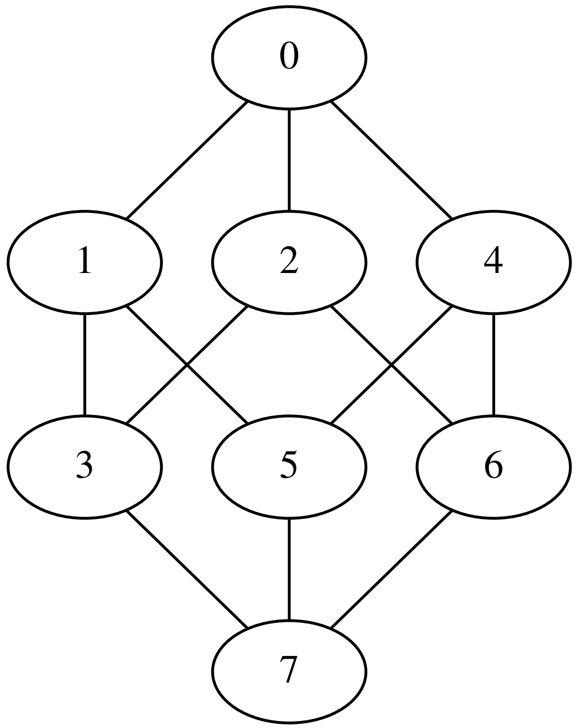
- Maintain *n* pSync arrays
- Each collective locks and uses a pSync from the pool of free arrays
- On barrier, unlock all arrays except for the current barrier's array
- If free arrays depleted, wait or allocate more

How to Reuse pSync: Strategy 3 (Pairwise Synchronisation)

- No pSync arrays - synchronisation is pairwise and memory is exclusive
  - Each target PE needs a separate piece of memory to synchronise on
- Maintain multiple memory locations for each target to rotate through (similar to before)
- Memory use dependent on synchronisation algorithm(s) used
  - Tree and recursive doubling algorithms are usually sufficient
- Whenever a PE needs to synchronise with another as a part of the chosen algorithm, it selects and locks one of the dedicated locations for that other PE
- Receipt of a synchronisation message from another PE unlocks all buffers for that PE

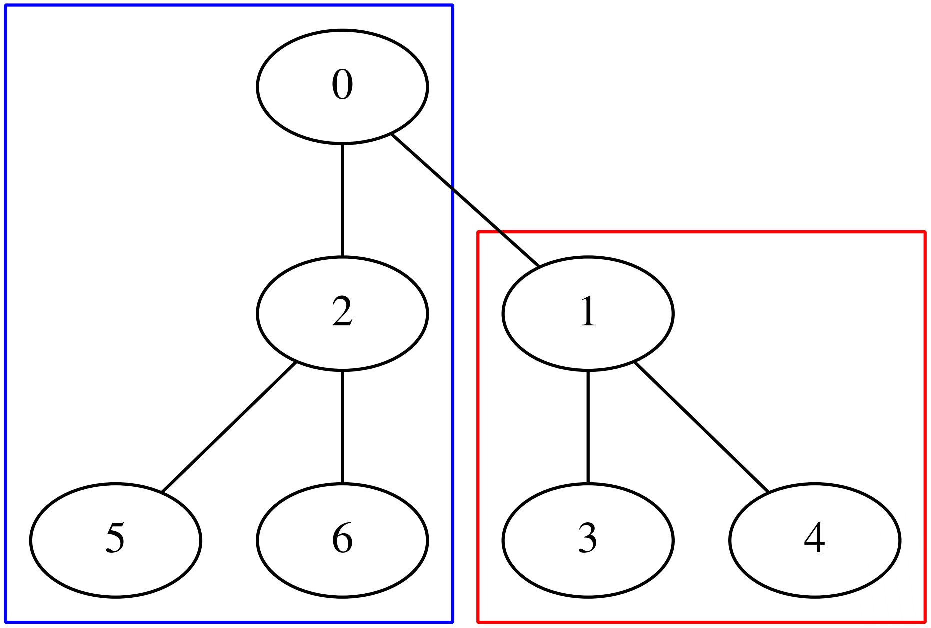
OAK RIDGE
National Laboratory

Strategy 3: Recursive doubling



- Edges always remain the same

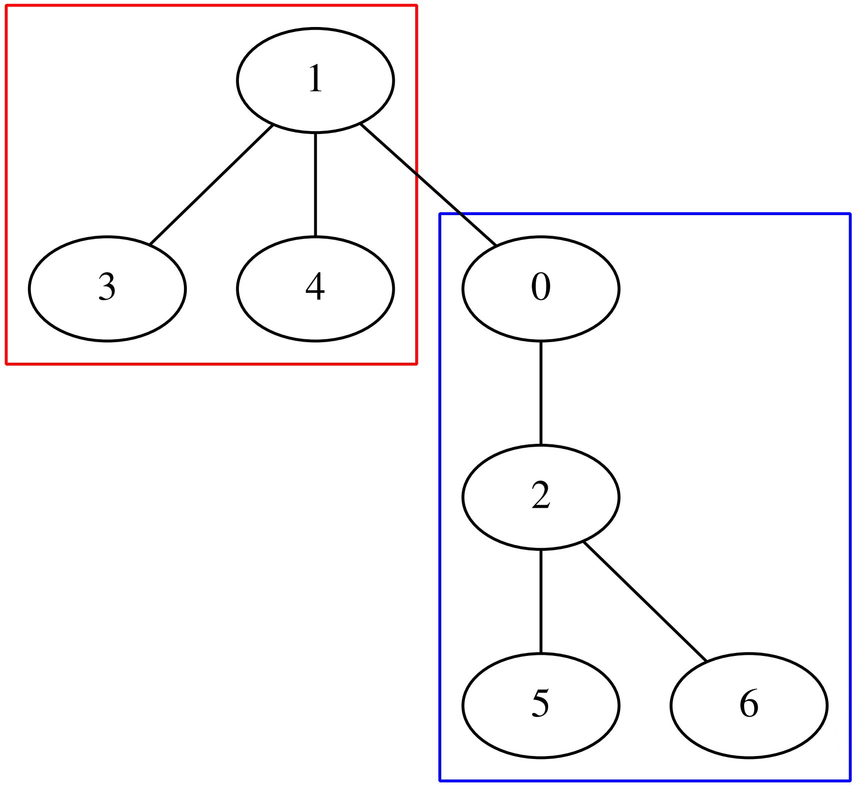
Strategy 3: Tree (Fixed Structure)



- Maintaining balanced tree results in edges changing based on root

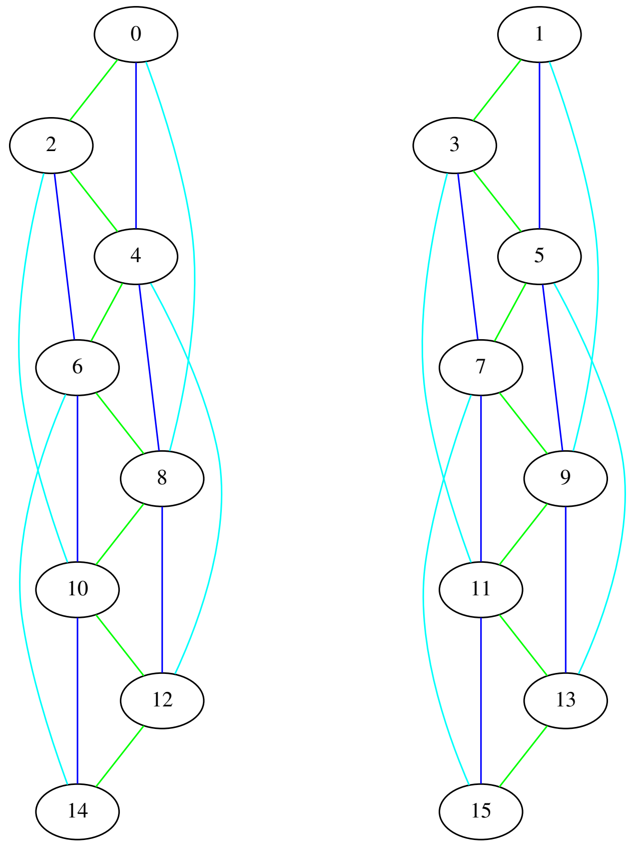
OAK RIDGE National Laboratory

Strategy 3: Tree (Fixed Edges)

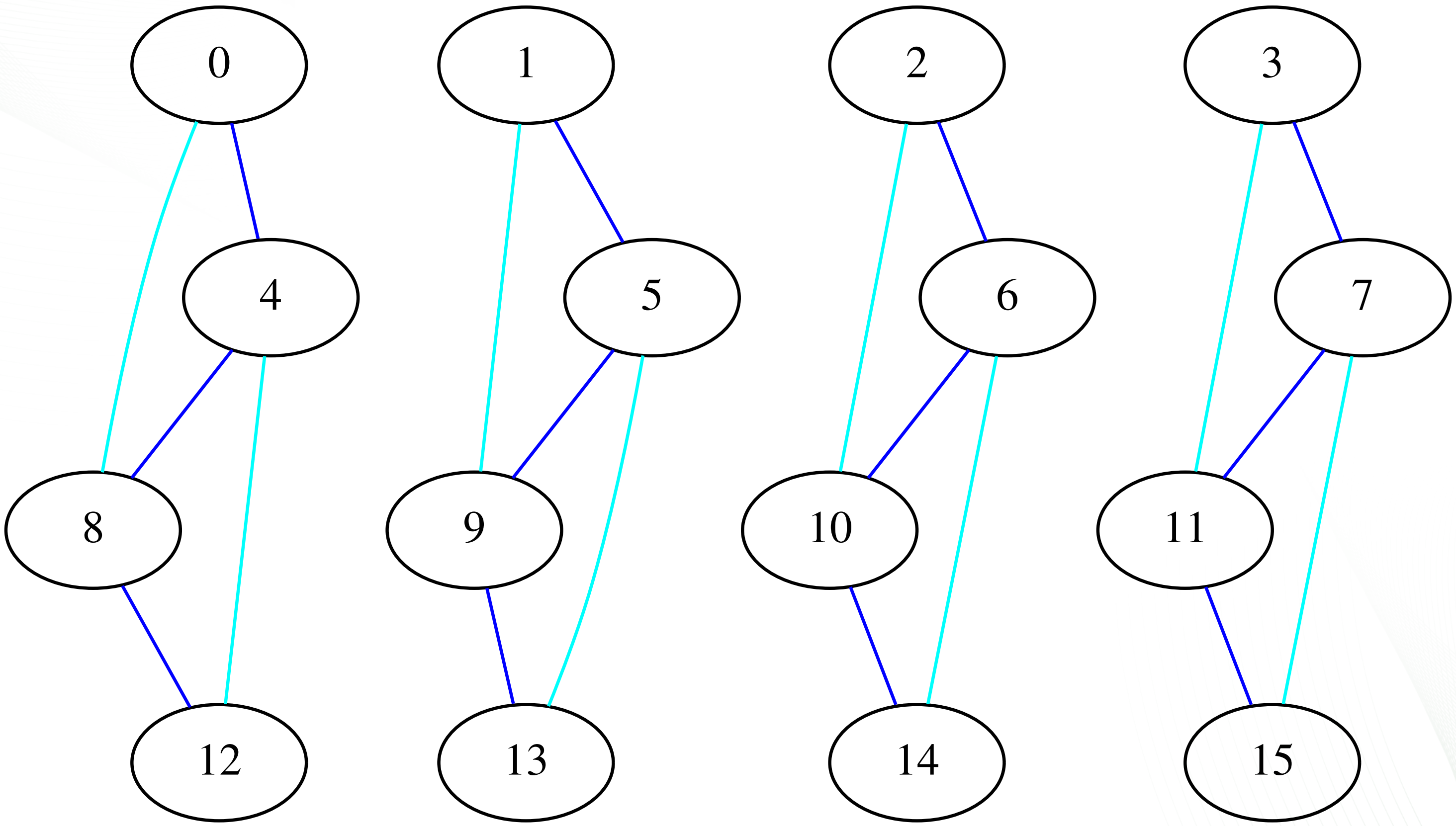


- Maintaining edges results in unbalanced tree

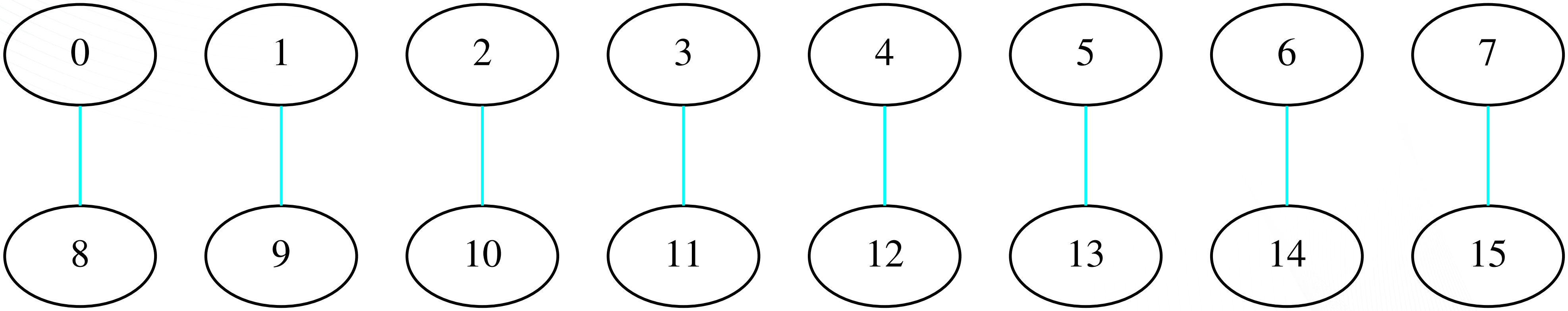
Connectivity Graph: Stride 2



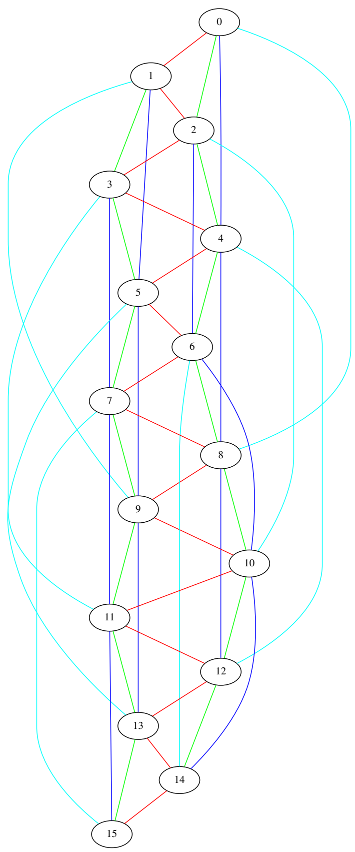
Connectivity Graph: Stride 4



Connectivity Graph: Stride 8



Connectivity Graph: All Strides



OAK RIDGE National Laboratory

Theoretical Analysis

- Strategy 1 - Additional Synchronisation
  - Communication cost: $\mathcal{O}(3 \log n)$ (worst case)
  - Memory Cost: $\mathcal{O}(n \log n)$
- Strategy 2 - Unlock on User Barrier
  - Communication cost: $\mathcal{O}(2 \log n)$ (worst case)
  - Memory cost: $\mathcal{O}(cn \log n)$, where c is the average distance (in collective operations) between barriers

OAK RIDGE National Laboratory

Theoretical Analysis

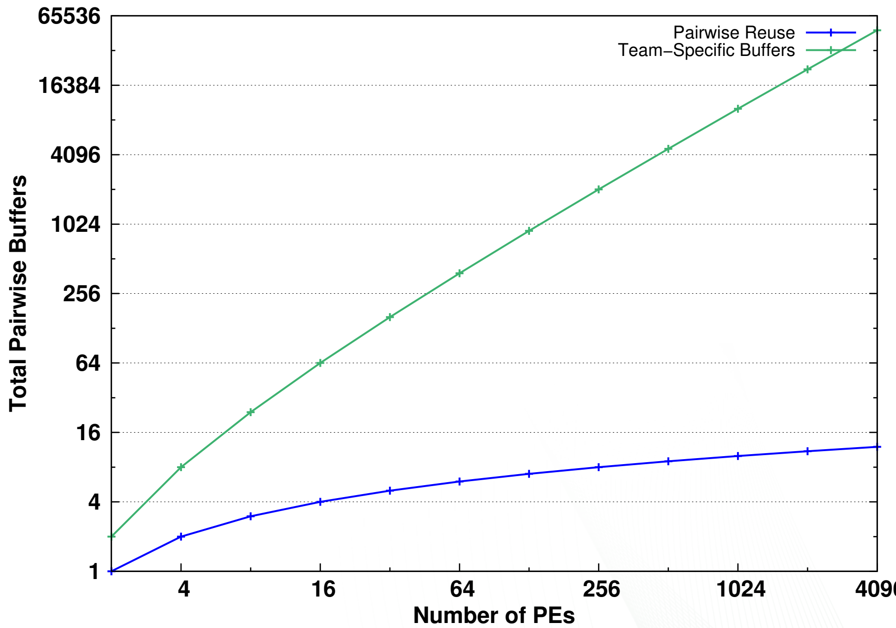
- Strategy 3 - Pairwise Synchronisation
  - Recursive doubling
    - Communication cost: $\mathcal{O}(\log n)$
    - Memory cost: $\mathcal{O}(n \log n)$
  - Tree (fixed structure)
    - Communication cost: $\mathcal{O}(\log n)$
    - Memory cost: $\mathcal{O}(n^2)$
  - Tree (fixed edges)
    - Communication cost: $\mathcal{O}(2 \log n)$
    - Memory cost: $\mathcal{O}(3)$

SHOC Benchmark Suite: QTC

- Scalable Heterogeneous Computing (SHOC) benchmark suite
  - Collection of benchmark programs testing the performance and stability of systems using computing devices with non-traditional architectures for general purpose computing, and the software used to program them
  - Initial focus is on systems containing graphics processing units (GPUs) and multi-core processors
- SHOC: quality threshold clustering (QTC)
  - Clustering algorithm like *k*-means
  - Instead of finding points near *k* centroids, group points based on some distance threshold
    - Variable number of clusters
  - As remaining free points decreases, excess PEs are pruned from working group

QTC Results

- Memory use for pairwise strategy
- Memory requirement: log *n*
- Memory use with unique, traditional pSync arrays for each *team*: *n* log *n*



OAK RIDGE National Laboratory

Conclusion

- Moving synchronisation buffer management to implementation shown to be possible with set degrees of overhead
- Possible to scale linearly with respect to the number of PEs in the system
- May involve making decisions on acceptable tradeoffs

Future Work

- Investigate when it may make sense to destroy old buffers
- Consider implications and potential benefits for applying tags to collective operations
- Analyse other potential synchronisation algorithms that may be used
- Further study optimal connectivity graphs

Acknowledgements

This work is supported by the United States Department of Defense (DoD) and used resources of the Computational Research and Development Programs and the Oak Ridge Leadership Computing Facility (OLCF) at Oak Ridge National Laboratory.