Exploring Hybrid MPI+Kokkos Tasks Programming Model

Samuel Khuvis[1], Karen Tomko[1], Jahanzeb Hashmi[2], Dhabaleswar K. Panda[2]

[1]Ohio Supercomputer Center

[2]Department of Computer Science and Engineering, The Ohio State University

Outline

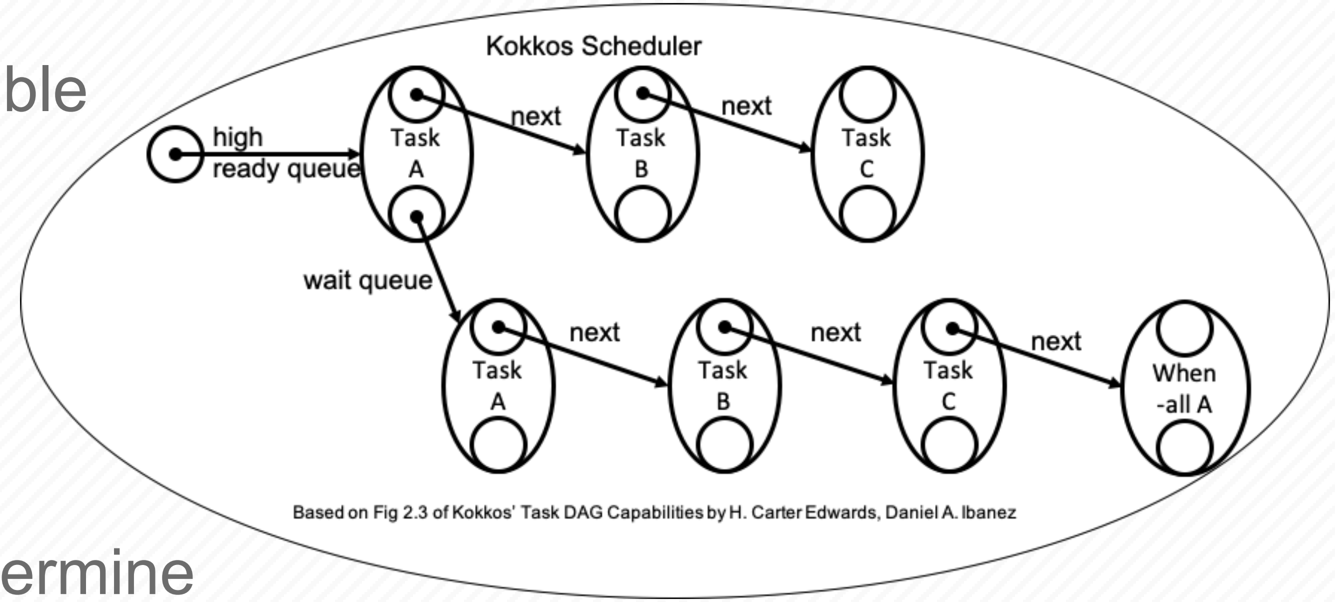
- *Introduction*
- *GEMM*
  - *Implementation*
  - *Results*
- *Graph500*
  - *Implementation*
  - *Results*
- *Limitations*
- *Summary*

Motivation

- Most current parallel implementations use data parallelism (MPI/MPI+OpenMP).
- Recent studies show that these models may not be well suited for emerging application.
- Task-based parallelism has potential to perform better at large-scale.
- Well-implemented task-based code will provide additional work to resources that would be otherwise underutilized.
- Most legacy scientific codes use MPI, so viable approach is to port sections of code to new models (i.e., PGAS or tasks).

Background

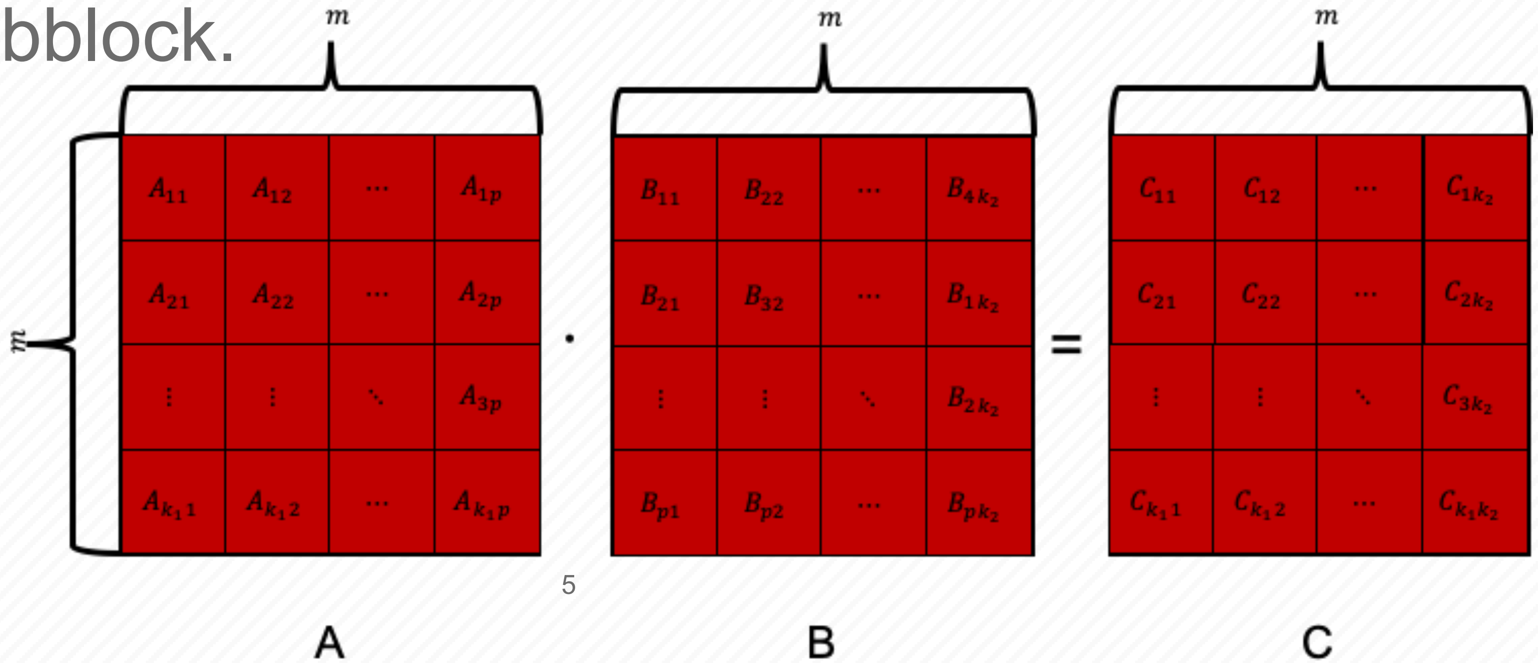
- Kokkos is a C++ library for writing portable applications across different computing architectures.
- Uses a shared memory model.
- Allows for a task-based approach to parallelization.
- Tasks enqueued to a scheduler.
- Future is generated by scheduler to determine when a task has completed.
- Futures can also be used to define dependences between tasks
- A when-all entity is a future that completes when multiple tasks have completed.



Based on Fig 2.3 of Kokkos' Task DAG Capabilities by H. Carter Edwards, Daniel A. Ibanez

Kokkos GEMM Implementation

- Uses tasks for shared memory parallelism.
- Main function spawns 1 task with a triply nested for loop.
- Each iteration of loop launches one task at a time on each subblock of the matrices to compute the product on that subblock.

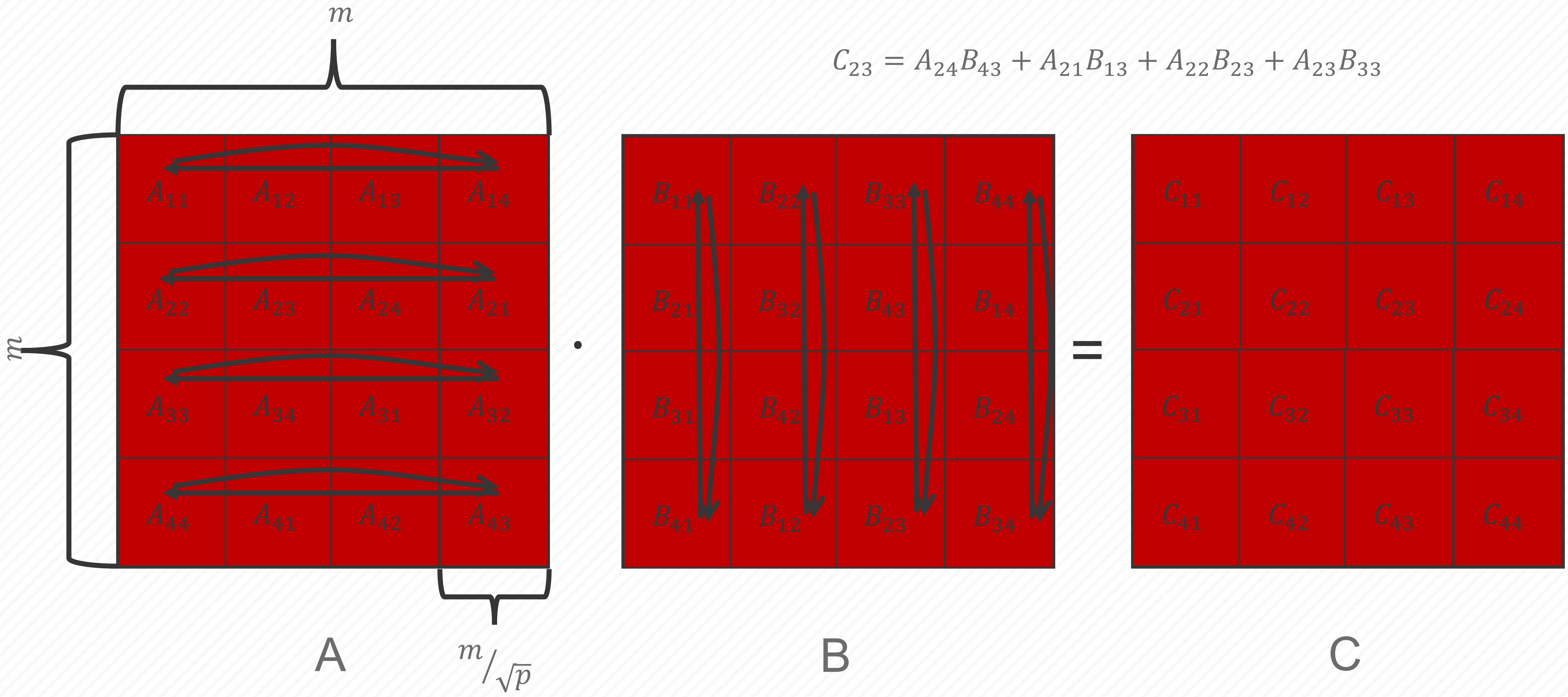

m
m
m
m
A_{11} A_{12} $\cdots$ A_{1p}
A_{21} A_{22} $\cdots$ A_{2p}
$\vdots$ $\vdots$ $\ddots$ A_{3p}
$A_{k_1 1}$ $A_{k_1 2}$ $\cdots$ $A_{k_1 p}$
$\cdot$
B_{11} B_{22} $\cdots$ B_{4k_2}
B_{21} B_{32} $\cdots$ B_{1k_2}
$\vdots$ $\vdots$ $\ddots$ B_{2k_2}
B_{p1} B_{p2} $\cdots$ B_{pk_2}
$=$
C_{11} C_{12} $\cdots$ C_{1k_2}
C_{21} C_{22} $\cdots$ C_{2k_2}
$\vdots$ $\vdots$ $\ddots$ C_{3k_2}
$C_{k_1 1}$ $C_{k_1 2}$ $\cdots$ $C_{k_1 k_2}$
A
B
C

MPI+Kokkos GEMM Implementation

- Add MPI in order to scale on multiple nodes.
- Use Cannon’s algorithm since it scaled well and keeps memory usage constant as the number of ranks grows.
- No modifications to tasks since each GEMM task performs computation on local subblocks.

Cannon's Algorithm on $p = 16$ processes

$$C_{23} = A_{24}B_{43} + A_{21}B_{13} + A_{22}B_{23} + A_{23}B_{33}$$



A_{11}	A_{12}	A_{13}	A_{14}
A_{22}	A_{23}	A_{24}	A_{21}
A_{33}	A_{34}	A_{31}	A_{32}
A_{44}	A_{41}	A_{42}	A_{43}

B_{11}	B_{22}	B_{33}	B_{44}
B_{21}	B_{32}	B_{43}	B_{14}
B_{31}	B_{42}	B_{13}	B_{24}
B_{41}	B_{12}	B_{23}	B_{34}

C_{11}	C_{12}	C_{13}	C_{14}
C_{21}	C_{22}	C_{23}	C_{24}
C_{31}	C_{32}	C_{33}	C_{34}
C_{41}	C_{42}	C_{43}	C_{44}

GEMM Code on 1 node

Kokkos Tasks	Runtime (in sec)	Speedup	Efficiency
1	1223	1	1
2	681	1.80	0.90
4	336	3.64	0.91
8	174	7.02	0.88
16	97	12.61	0.79
32	58	21.09	0.66
40	50	24.46	0.61

- Problem size of 20,000 and block size of 256.
- Run on 1 rank.
- Parallel efficiency is below ideal efficiency.

GEMM Code over multiple nodes

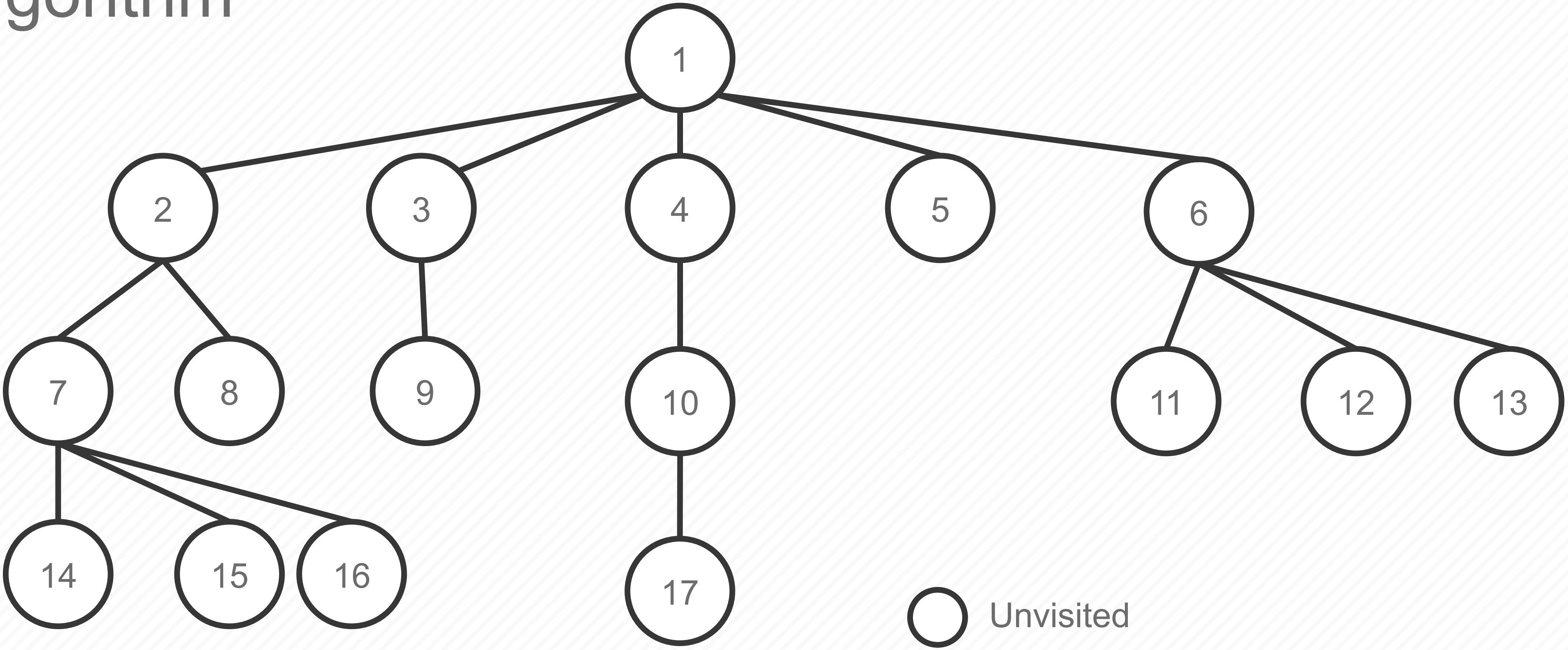
Nodes	Runtime (in sec)	Speedup	Efficiency
9	72.5	1	1
16	43.5	1.67	0.94
25	27.7	2.62	0.94
36	20.4	3.55	0.89
64	11.8	6.15	0.87

- Problem size of 45,000 and block size of 256.
- Run with 1 rank/node and 40 tasks/rank.
- Adding MPI enables larger problems to be solved than Kokkos-only.
- Scales well up to 64 nodes (or 2560 tasks).

MPI Graph500 Implementation

- Breadth first search (BFS) function is passed graph, root vertex, and visited array.
- Key data structures:
  - `oldq`: array of vertices visited in previous iteration,
  - `newq`: array of neighbors of vertices in oldq,
  - `visited`: array of vertices that have been visited,
  - `pred`: array that maps neighbor vertex to source vertex.

BFS Algorithm



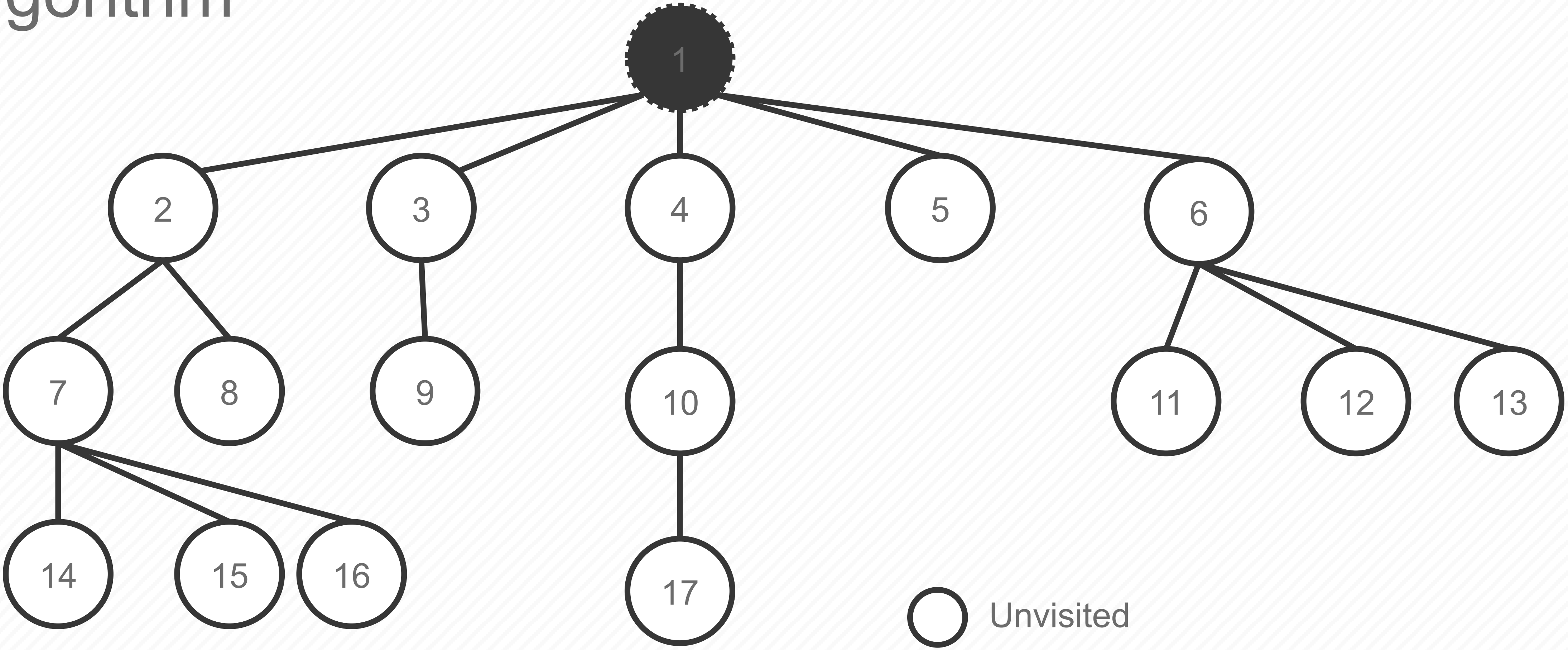
○ Unvisited

◍ Neighbors of nodes in oldq, to be added to newq

● Nodes in visited

◉ Nodes in oldq and visited

BFS Algorithm



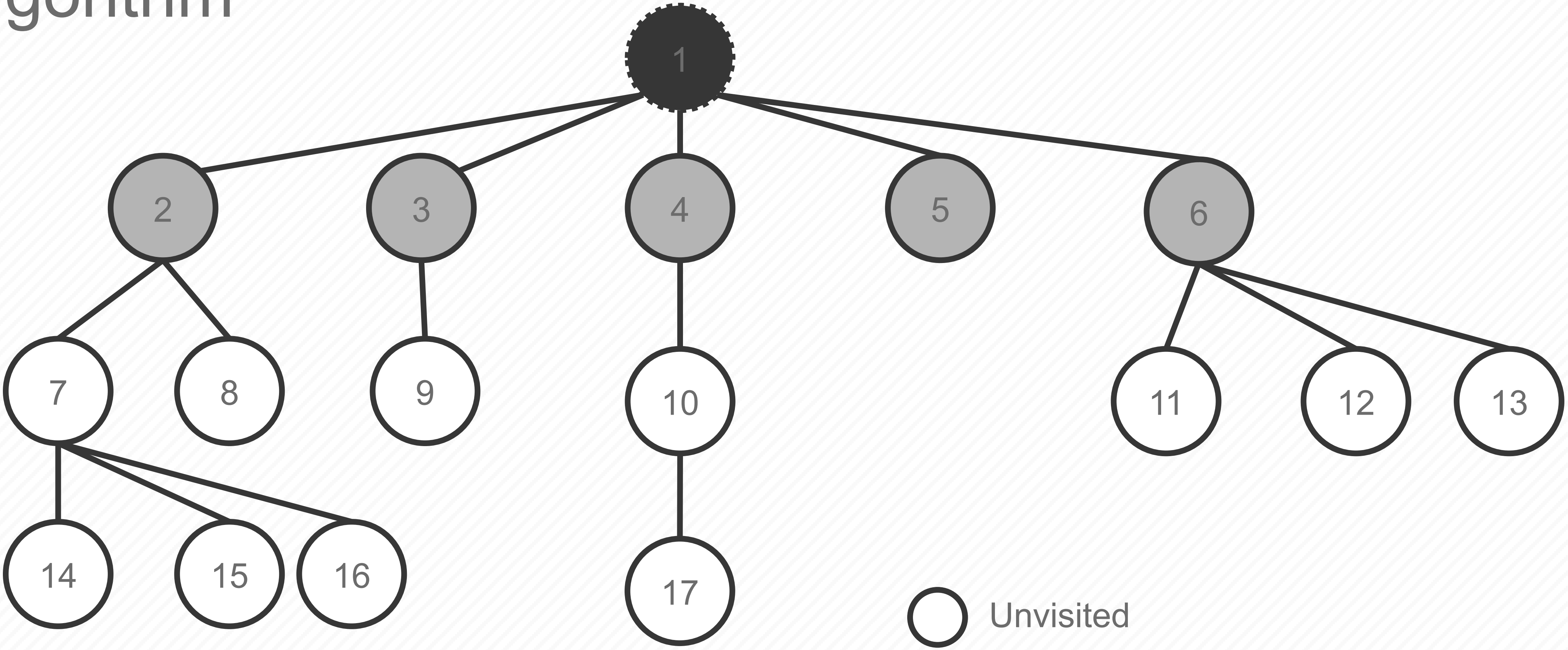
○ Unvisited

◍ Neighbors of nodes in oldq,
to be added to newq

● Nodes in visited

◌ Nodes in oldq and visited

BFS Algorithm



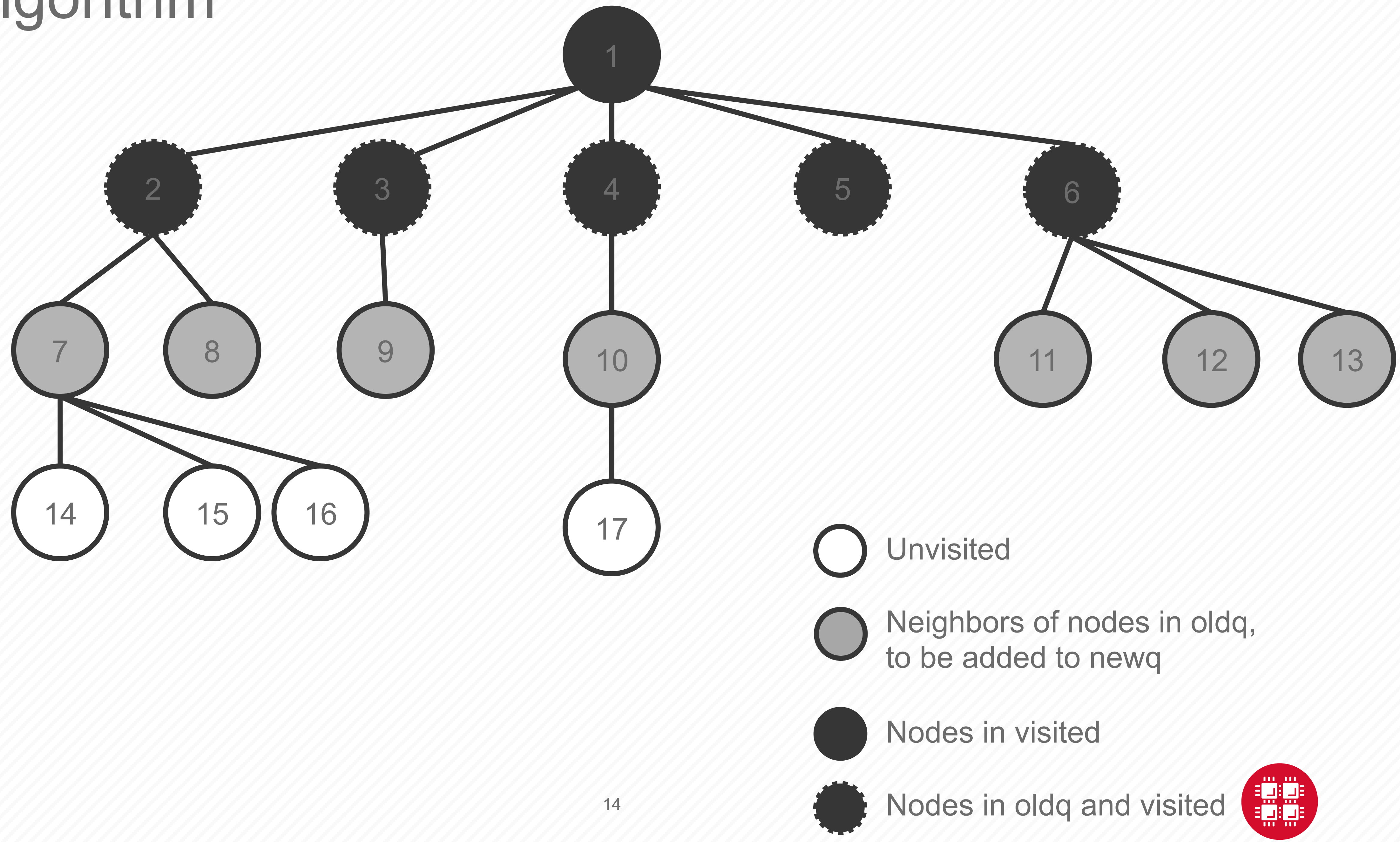
○ Unvisited

◍ Neighbors of nodes in oldq,
to be added to newq

● Nodes in visited

⬤ Nodes in oldq and visited

BFS Algorithm



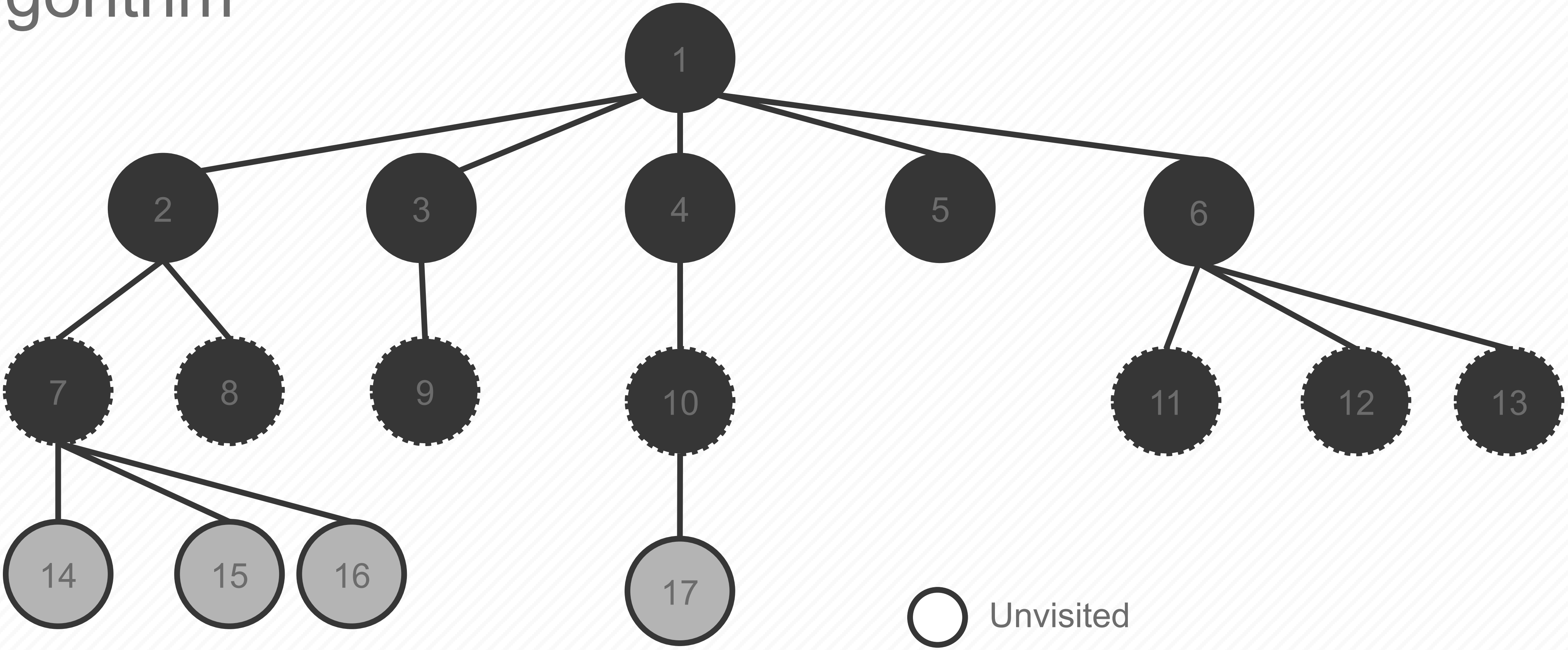
○ Unvisited

◍ Neighbors of nodes in oldq, to be added to newq

● Nodes in visited

◌ Nodes in oldq and visited

BFS Algorithm



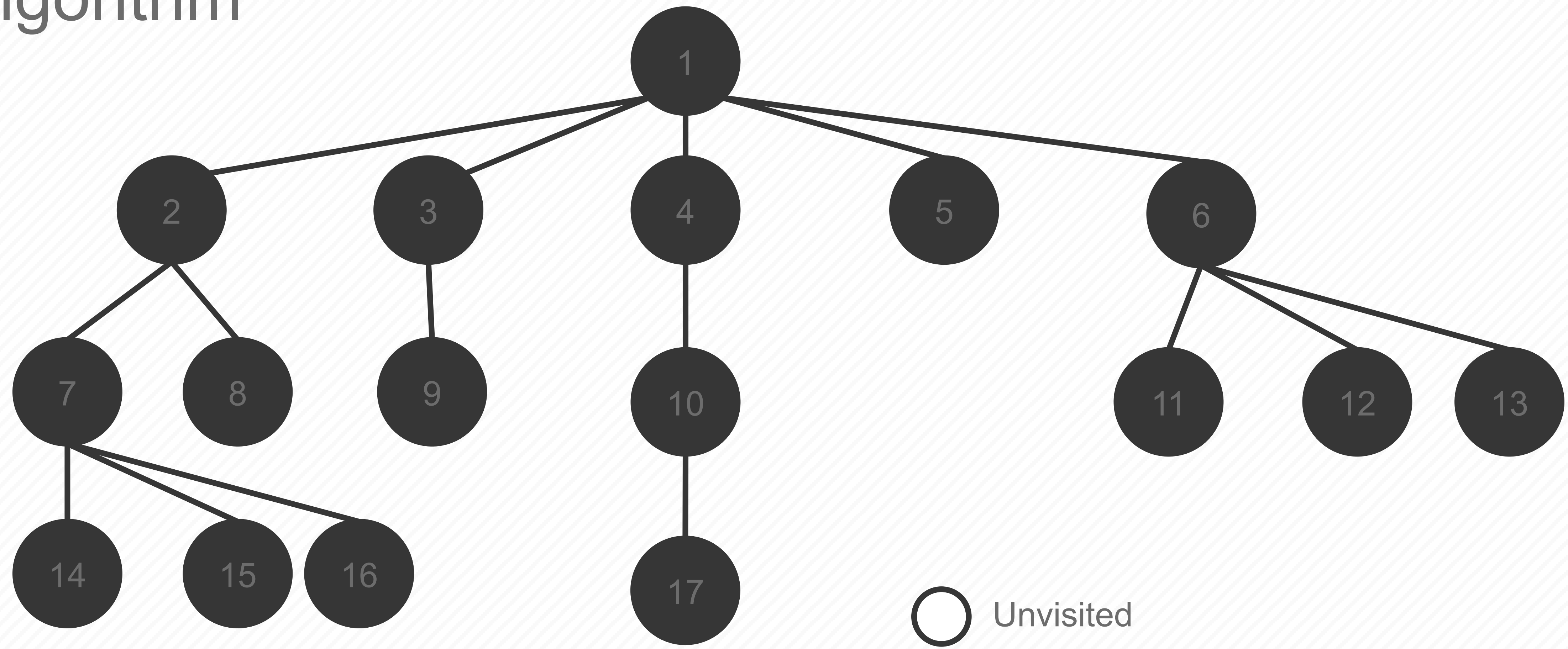
○ Unvisited

● Neighbors of nodes in oldq, to be added to newq

● Nodes in visited

● Nodes in oldq and visited

BFS Algorithm



○ Unvisited

◍ Neighbors of nodes in oldq, to be added to newq

● Nodes in visited

◌ Nodes in oldq and visited

MPI+Kokkos Graph500 Implementation (locking)

- Use MPI implementation as starting point.
- Each process spawns NT tasks:
  - $NT - 1$ processing tasks that iterate over a subset of `oldq`,
  - 1 receive task that waits for data from other ranks and processes it as it receives it. Runs until all processing tasks send a signal that they have completed.
- All tasks modify `visited`, `pred`, and newq arrays so updates are locked to avoid race conditions.
- `newq_count` atomic variable used to track elements in `newq`.

Locking Graph500 Results

Tasks	BFS	Speedup	Efficiency	Locks
2	48.7	2.0	1.00	3.78
4	106.4	0.92	0.23	53.01
8	141.2	0.69	0.086	88.58
16	162.0	0.60	0.038	92.49
32	248.1	0.39	0.012	161.96
40	259.2	0.38	0.0094	162.85

- SCALE=20.
- Running on 1 MPI process.
- Increasing number of tasks increases runtime.
- Time spent in locks also increases with number of tasks.
- Significant percentage of slowdown caused by time spent in locks.

MPI+Kokkos Graph500 Implementation (non-locking)

- Removes all locks and atomics.
- Each task maintains its local version of `visited`, `pred`, and `newq` arrays.
- Once all tasks have completed, local arrays from all tasks on a rank are merged into global arrays.

Non-locking Graph500 Results

Tasks	BFS	Speedup	Efficiency
2	25.5	2.0	1.00
4	18.7	2.7	0.68
8	14.2	3.6	0.45
16	11.3	4.5	0.28
32	10.6	4.8	0.15
40	10.2	5.0	0.13

- Better absolute runtimes than locking implementation.
- Scales better than locking implementation, but still poorly.

Non-locking Graph500 Results

Processes	Processing Tasks	Runtime (in s)
1	16	11.6
2	8	11.1
4	4	7.5
8	2	7.6
16	1	4.7

- Test different combinations processes and tasks for a total of 16 processing tasks.
- Increasing processes/ decreasing tasks improves performance.
- Best performance with 16 processes and 1 processing task.

Imbalance Percentage for Non-locking Graph500 Results

Tasks	Max Imbalance	Mean Imbalance
4	0.857	0.471
8	0.946	0.471
16	0.956	0.472
32	0.926	0.476
40	0.956	0.572

- Percentage of time task spends waiting for other tasks to complete.
- For each iteration, compute difference in runtime between slowest and fastest tasks, then divide by time spent in slowest task.
- On average, over all iterations, ~50% of task time is spent in waiting.
- For some iterations, imbalance can be as high as 96%.

Graph500 Results Comparison

Processes	MPI	Locking MPI+Kokkos	Non-locking MPI+Kokkos
1	10.5	254.5	10.2
2	13.4	83.2	12.3
4	7.5	34.1	7.8
8	4.3	15.0	5.8
16	2.9	7.4	4.4
32	2.3	N/A	N/A

- Non-locking performs better than locking.
- Best performance with MPI-only.

Limitations

- Profiles of Graph500 show more time is spent in MPI communication when using Kokkos (more data is sent with a lower call rate).
- Non-locking implementation of Graph500 performs additional work since multiple tasks may visit the same node.
  - For SCALE=20, 652,549 unique vertices but 2,515,974 to 4,077,491 vertices visited across all tasks (3.86 to 6.25 times more vertices).
- Can only check if all tasks have completed with a barrier.
  - ~50% of processing task time spent waiting.
  - Wait must complete for all tasks before merging local arrays into global arrays.
  - It would be beneficial to overlap queue processing in the tasks with merging local arrays by merging as tasks complete.

Summary

- Demonstrated feasibility of hybrid MPI+Kokkos tasks codes
- Demonstrated that hybrid MPI+Kokkos tasks model can efficiently solve larger problem sizes than Kokkos tasks alone
- Evaluated scaling of 2 example hybrid MPI+Kokkos codes
- Identified challenge to adopting this task model

Acknowledgements

- We would like to thank the Ohio Supercomputer Center for access to the Pitzer cluster.
- This research is supported in part by NSF grants #1931537 and #1565414.

OH·TECH

Ohio Technology Consortium
A Division of the **Ohio Department of Higher Education**

info@osc.edu

twitter.com/osc

facebook.com/ohiosupercomputercenter

osc.edu

oh-tech.org/blog

linkedin.com/company/ohio-supercomputer-center