Exploiting Thread Parallelism for Ocean Modeling on Cray XC Supercomputers

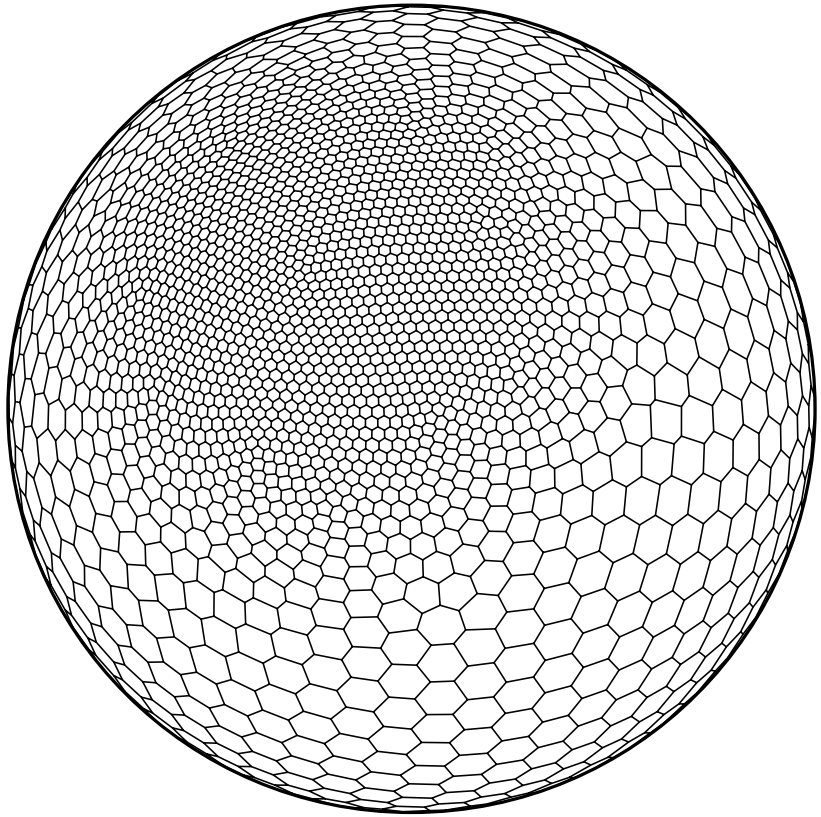
Abhinav Sarje
asarje @ lbl.gov

Cray Users Group Meeting (CUG)
May 2016

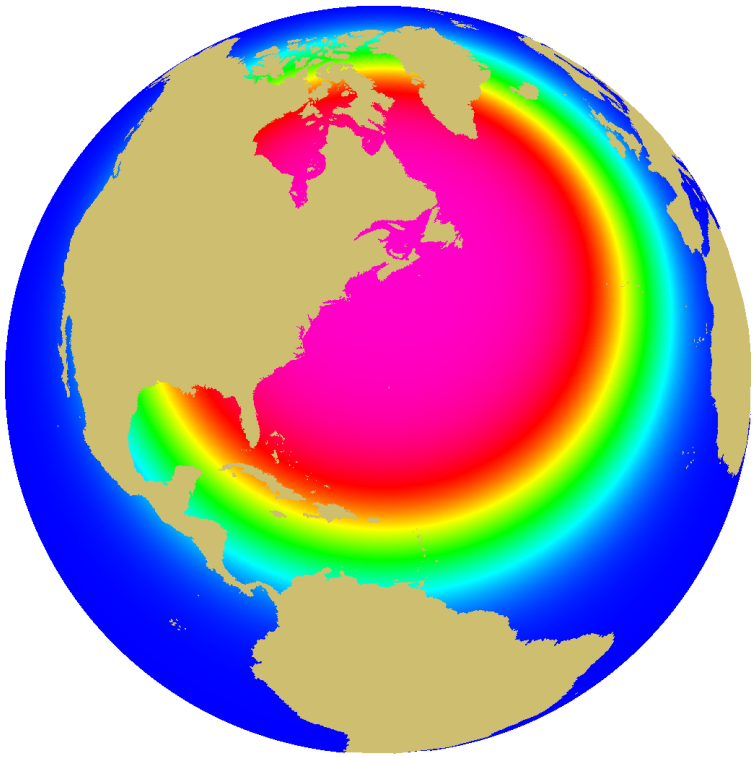
Abhinav Sarje	Lawrence Berkeley National Laboratory
Douglas Jacobsen	Los Alamos National Laboratory
Samuel Williams	Lawrence Berkeley National Laboratory
Todd Ringler	Los Alamos National Laboratory
Leonid Oliker	Lawrence Berkeley National Laboratory

Ocean Modeling with MPAS-Ocean

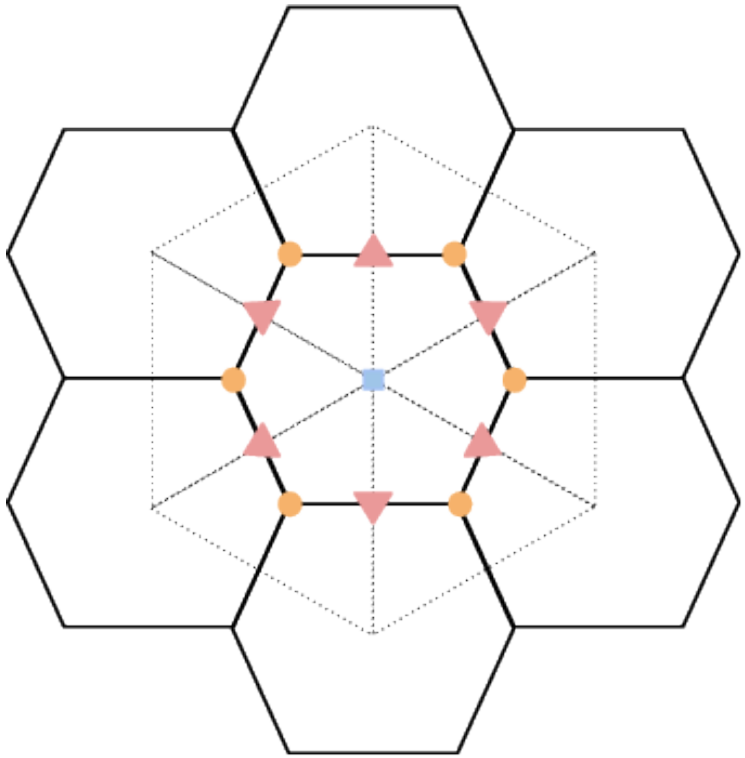
- MPAS = **M**odel for **P**rediction **A**cross **S**cales. [LANL/NCAR]
- A multiscale method for simulating Earth's oceans.
- 2D Voronoi tessellation-based variable resolution mesh (SCVT).
- Structured vertical columns in third dimension represent ocean depths.



Ocean Modeling with MPAS-Ocean



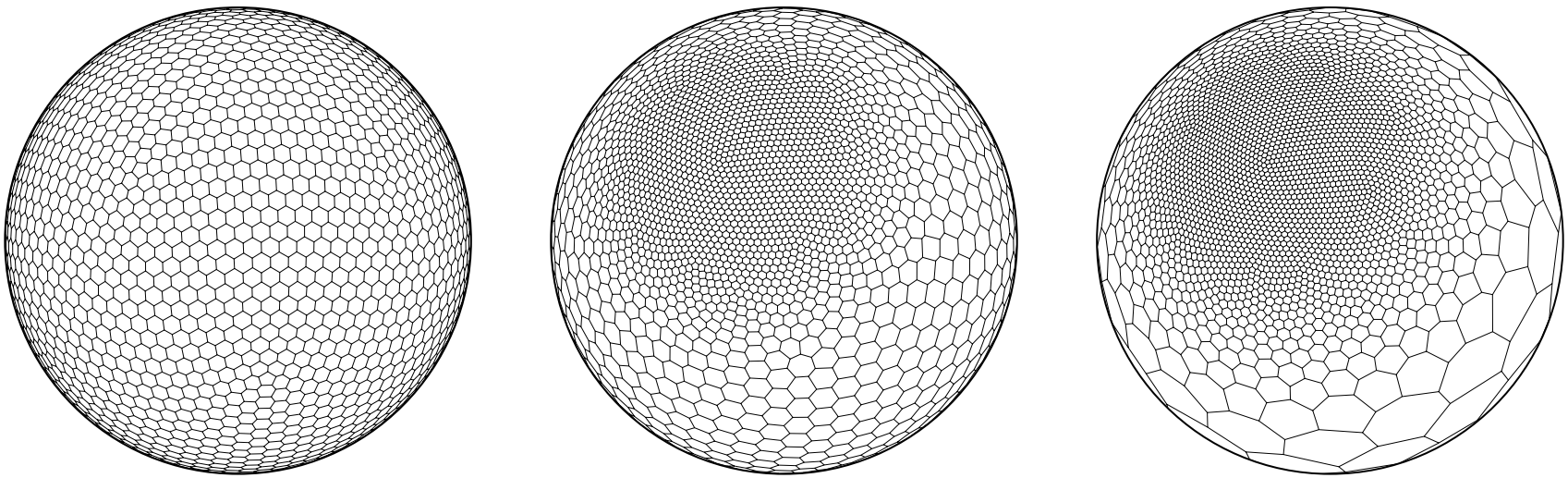
Variable Resolution Mesh



Mesh Elements

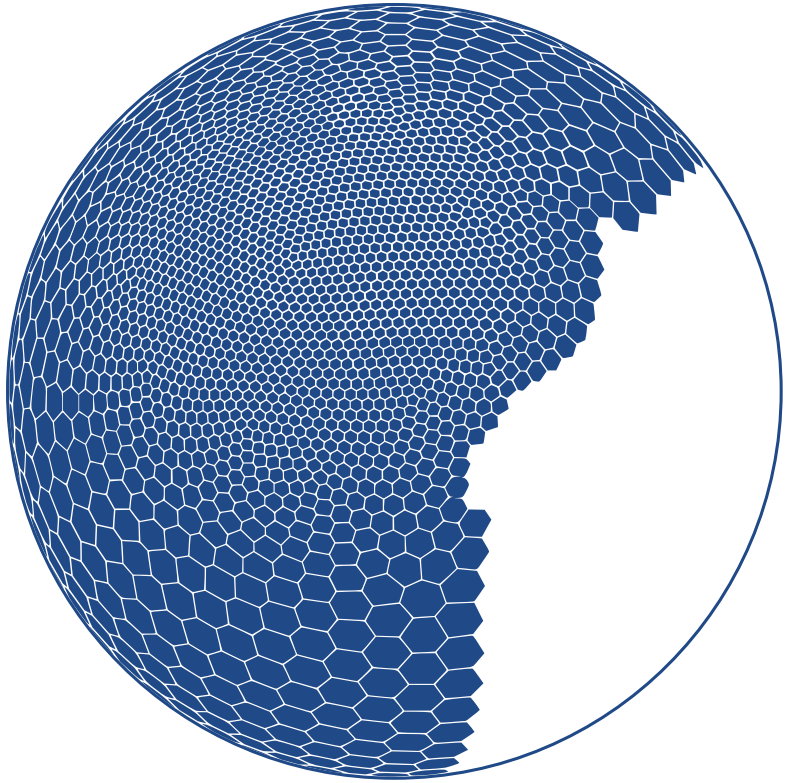
Ocean Modeling with MPAS-Ocean: Why Unstructured Meshes?

- Variable resolutions with any given density function.
- Straightforward mapping to flat 2D with effectively no distortions.
- Quasi-uniform (locally homogeneous) coverage of spherical surfaces.
- Smooth resolution transition regions.
- Preserve symmetry/isotropic nature of a spherical surface.
- Naturally allows unstructured discontinuities.



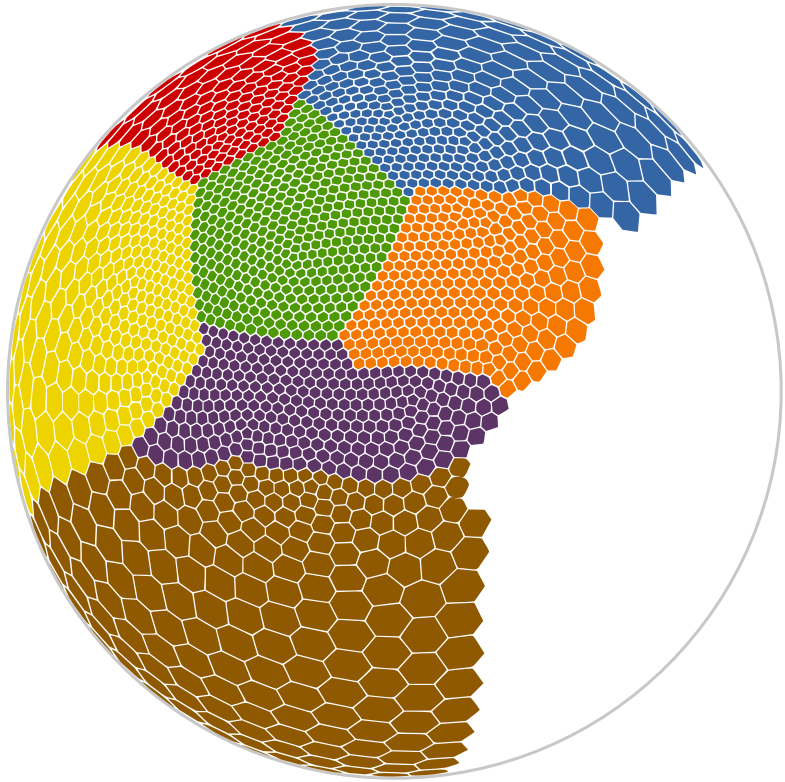
Background & Motivations

- Unstructured meshes in parallel environment.
- Challenging to achieve uniform partitioning unlike structured grids.
- Load balance depends on partitioning quality and element distributions.
- Data exchange between processes through deep *halo regions*.
- Threading motivated by increasing on-node core counts, with limited available memory per core.



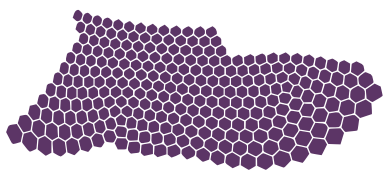
Background & Motivations

- Unstructured meshes in parallel environment.
- Challenging to achieve uniform partitioning unlike structured grids.
- Load balance depends on partitioning quality and element distributions.
- Data exchange between processes through deep *halo regions*.
- Threading motivated by increasing on-node core counts, with limited available memory per core.



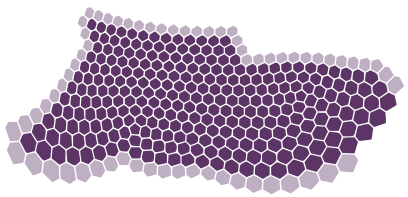
Background & Motivations

- Unstructured meshes in parallel environment.
- Challenging to achieve uniform partitioning unlike structured grids.
- Load balance depends on partitioning quality and element distributions.
- Data exchange between processes through deep *halo regions*.
- Threading motivated by increasing on-node core counts, with limited available memory per core.



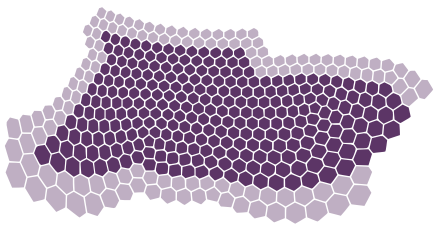
Background & Motivations

- Unstructured meshes in parallel environment.
- Challenging to achieve uniform partitioning unlike structured grids.
- Load balance depends on partitioning quality and element distributions.
- Data exchange between processes through deep *halo regions*.
- Threading motivated by increasing on-node core counts, with limited available memory per core.



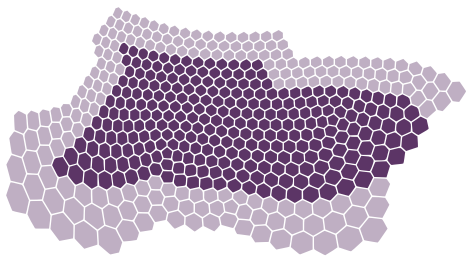
Background & Motivations

- Unstructured meshes in parallel environment.
- Challenging to achieve uniform partitioning unlike structured grids.
- Load balance depends on partitioning quality and element distributions.
- Data exchange between processes through deep *halo regions*.
- Threading motivated by increasing on-node core counts, with limited available memory per core.



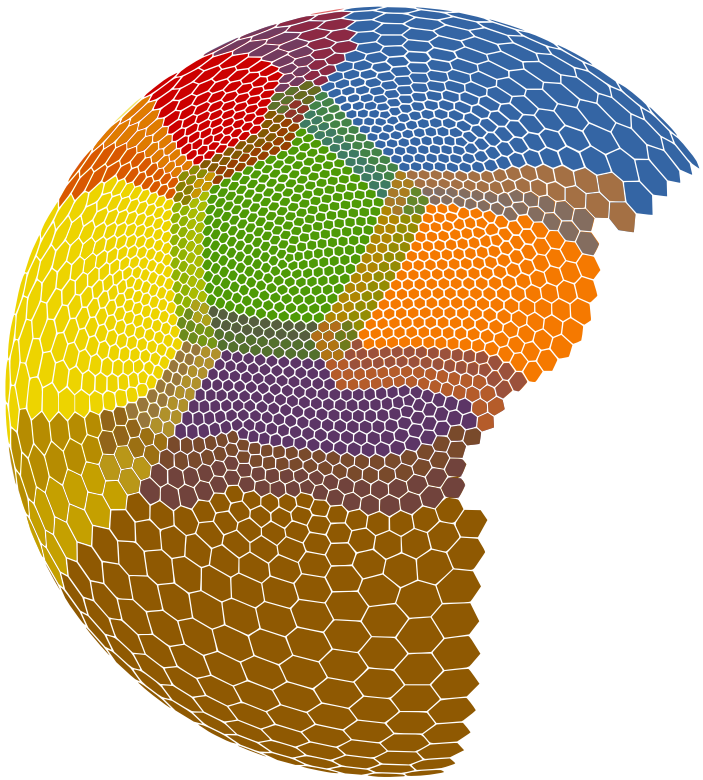
Background & Motivations

- Unstructured meshes in parallel environment.
- Challenging to achieve uniform partitioning unlike structured grids.
- Load balance depends on partitioning quality and element distributions.
- Data exchange between processes through deep *halo regions*.
- Threading motivated by increasing on-node core counts, with limited available memory per core.



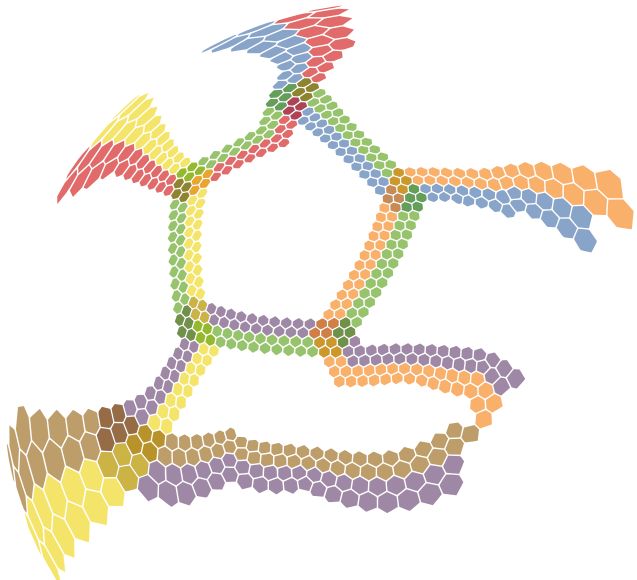
Background & Motivations

- Unstructured meshes in parallel environment.
- Challenging to achieve uniform partitioning unlike structured grids.
- Load balance depends on partitioning quality and element distributions.
- Data exchange between processes through deep *halo regions*.
- Threading motivated by increasing on-node core counts, with limited available memory per core.



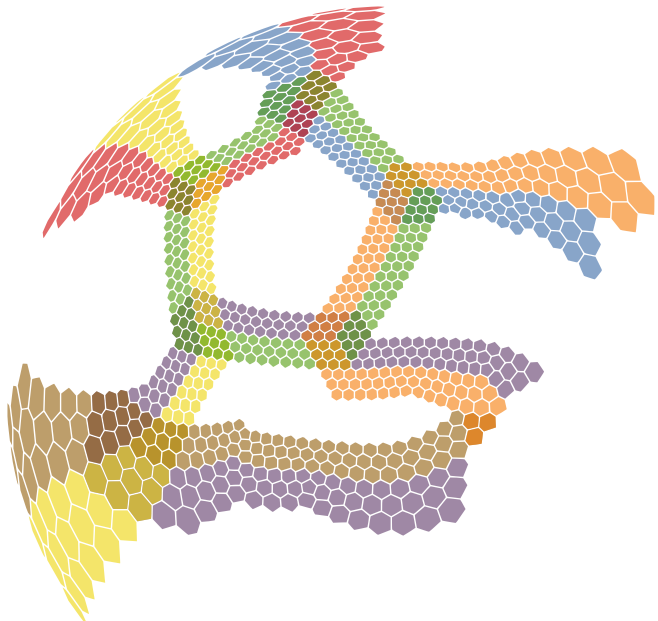
Background & Motivations

- Unstructured meshes in parallel environment.
- Challenging to achieve uniform partitioning unlike structured grids.
- Load balance depends on partitioning quality and element distributions.
- Data exchange between processes through deep *halo regions*.
- Threading motivated by increasing on-node core counts, with limited available memory per core.



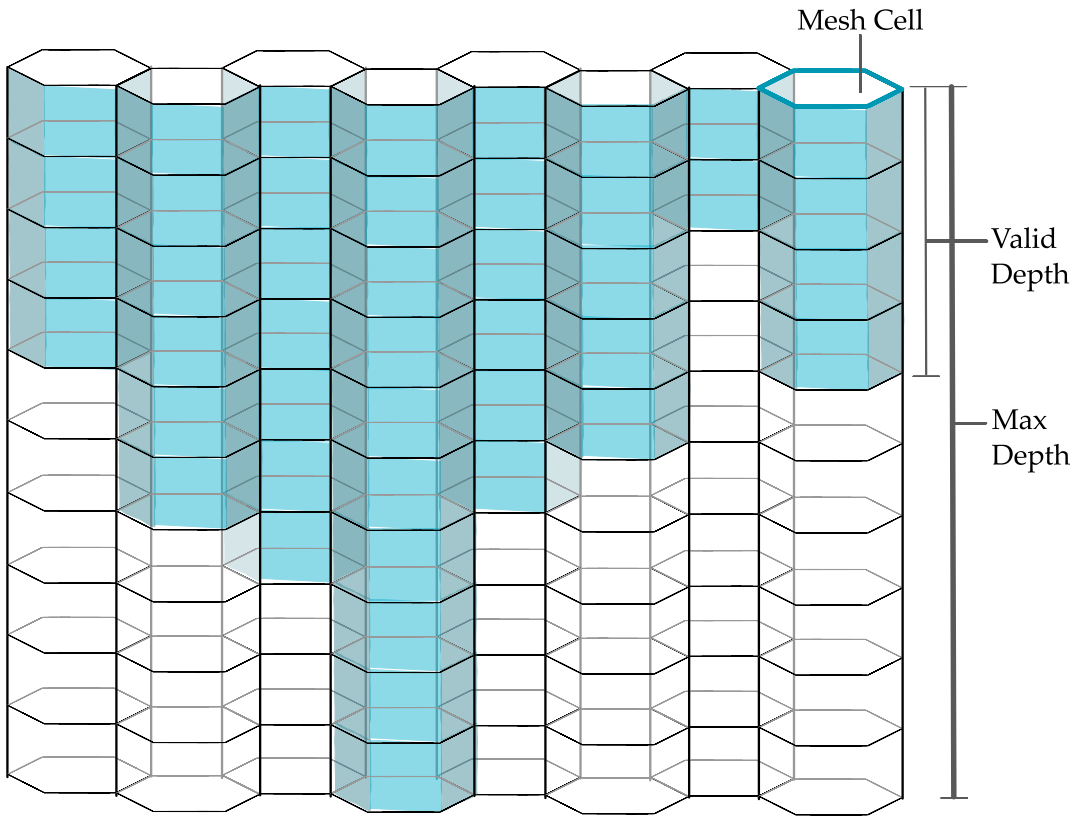
Background & Motivations

- Unstructured meshes in parallel environment.
- Challenging to achieve uniform partitioning unlike structured grids.
- Load balance depends on partitioning quality and element distributions.
- Data exchange between processes through deep *halo regions*.
- Threading motivated by increasing on-node core counts, with limited available memory per core.



Background & Motivations

- Structured vertical dimension for ocean depth.
- Variable depth but constant sized buffers with maximum depth!



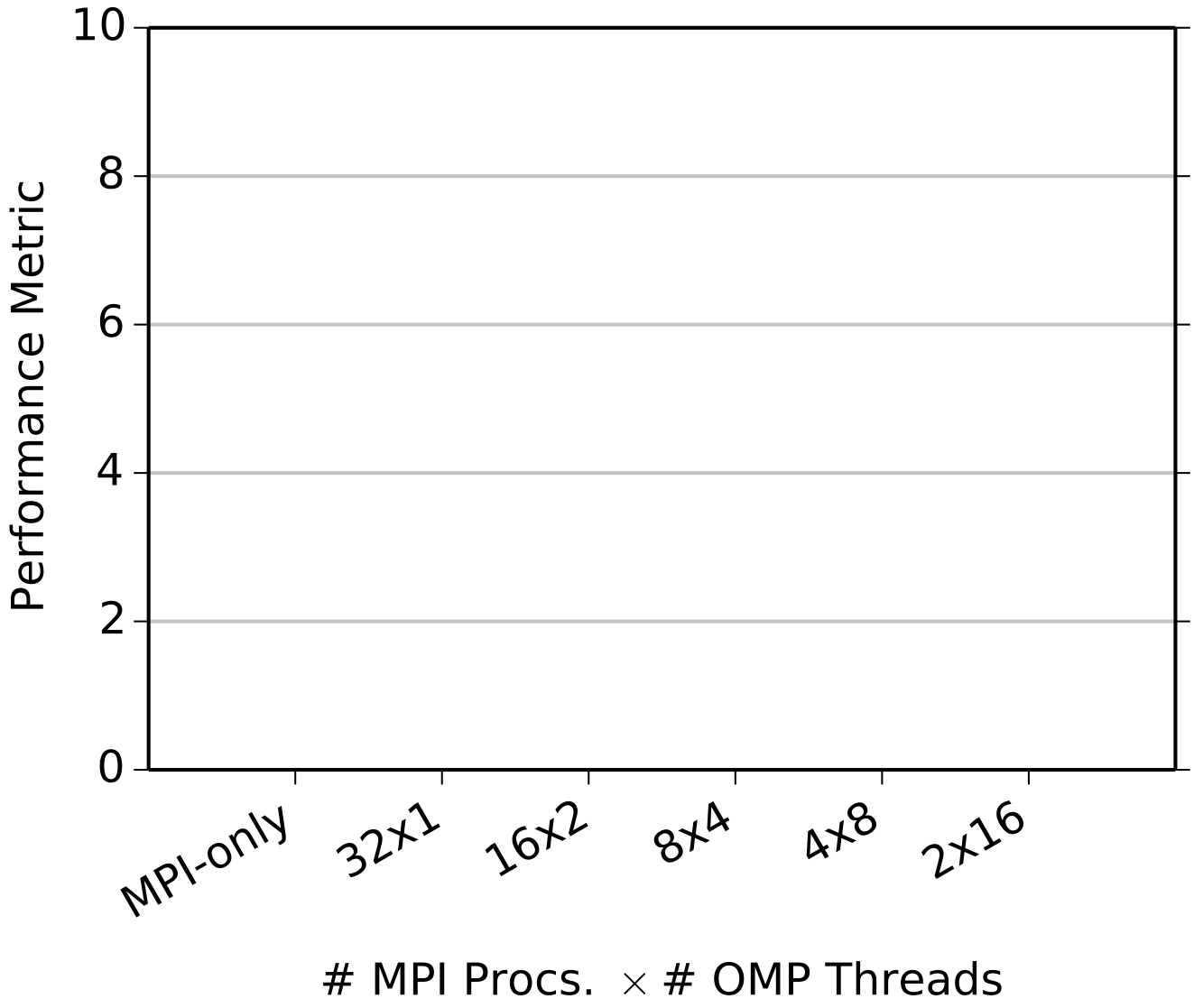
Developing a Hybrid MPI + OpenMP Implementation

Computational Environment

- Cray XC40: Cori Phase 1
  - 1,630 dual-socket compute nodes with Cray Aries interconnect
  - 16 core Intel Haswell processors
  - 32 cores per node
  - 128 GB DRAM per node

- Meshes:
  1. 60 kms uniform resolution:
     114,539 cells, maximum vertical depth of 40.
  2. 60 kms to 30 kms variable resolution:
     234,095 cells, maximum vertical depth of 60.

Performance Plots and Keys



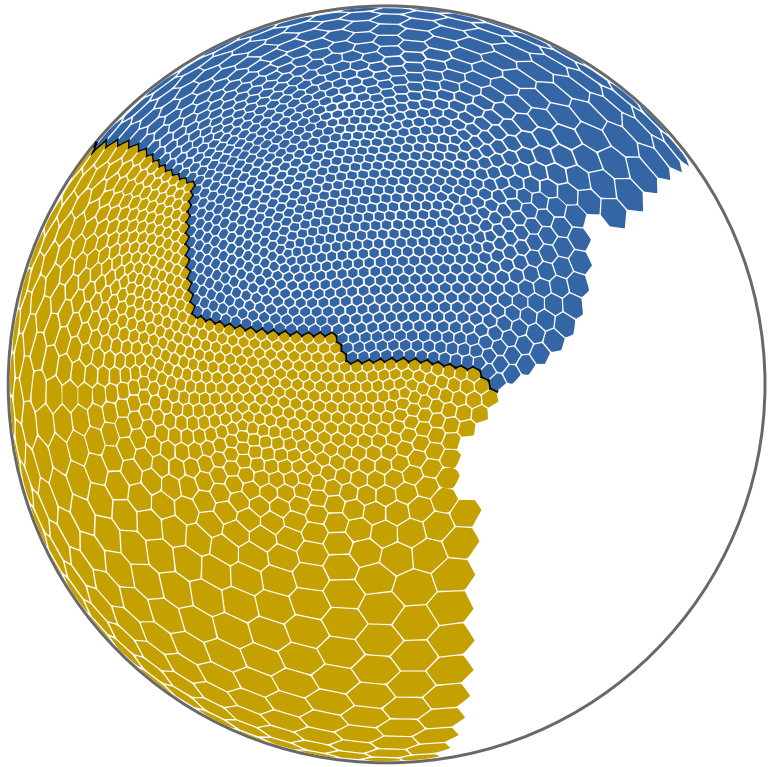
Performance Plots and Keys

Configurations = MPI procs. per node $\times$ OMP threads per MPI

Configuration	Decreasing	Increasing
1. 32×1	• Number of MPI processes per node	• Number of threads per process
2. 16×2	• Total halo region volume	• Amount of data sharing
3. 8×4	• Communication (number and volume)	• Threading overhead
4. 4×8	• *Extra* computations	• Inter-thread load imbalance
5. 2×16	• Memory requirements	
6. 1×32	• Inter-process load imbalance	

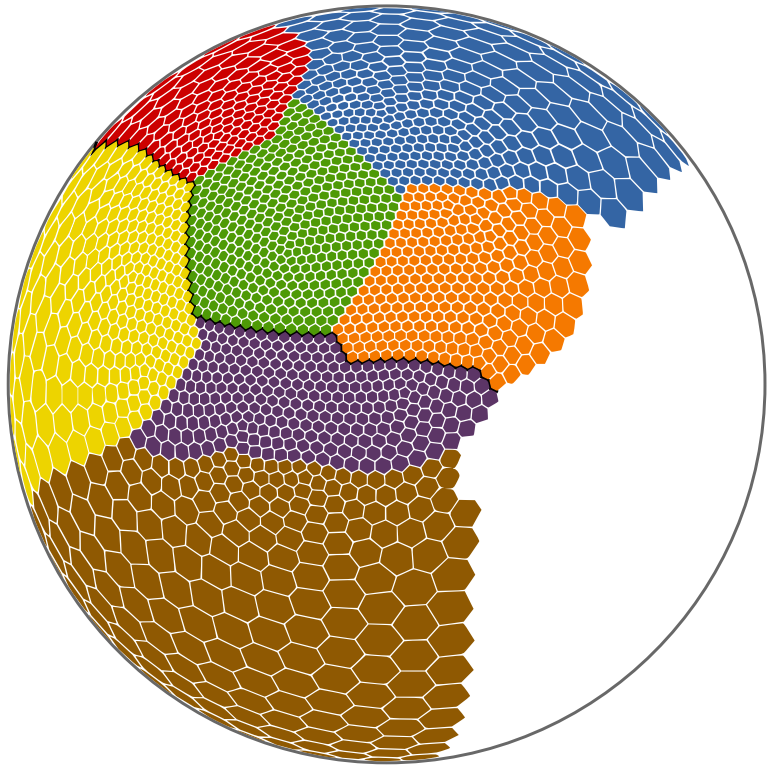
Threading Granularity: Block-Level

- Minimal implementation disruption to the code
- Simple intra-node halo-exchanges across threads without need for explicit communication
- Possibly insufficient parallelism
- Intra-block halo regions consume memory and require additional computations



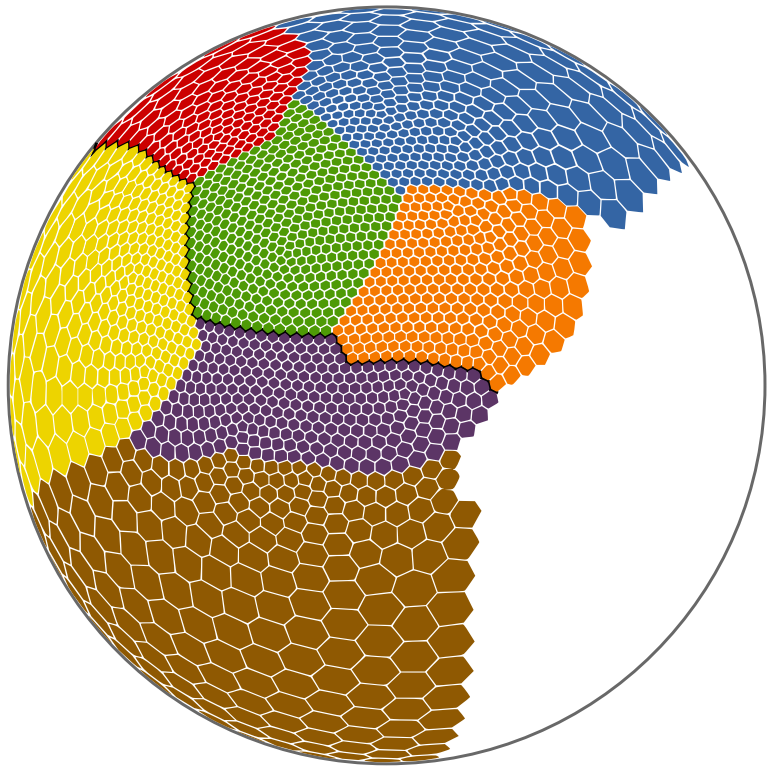
Threading Granularity: Block-Level

- Minimal implementation disruption to the code
- Simple intra-node halo-exchanges across threads without need for explicit communication
- Possibly insufficient parallelism
- Intra-block halo regions consume memory and require additional computations



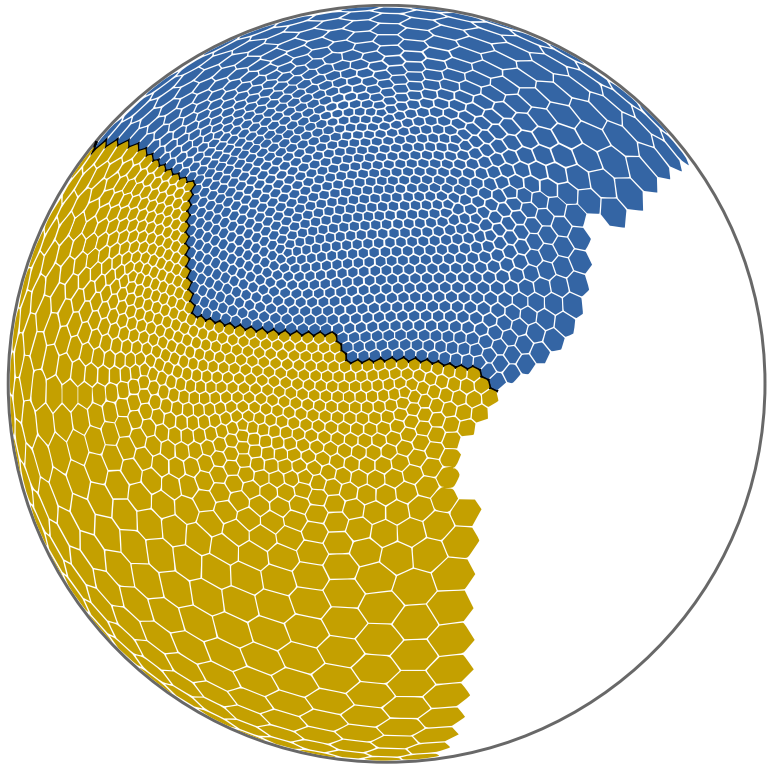
Threading Granularity: Block-Level

- Minimal implementation disruption to the code
- Simple intra-node halo-exchanges across threads without need for explicit communication
- Possibly insufficient parallelism
- Intra-block halo regions consume memory and require additional computations



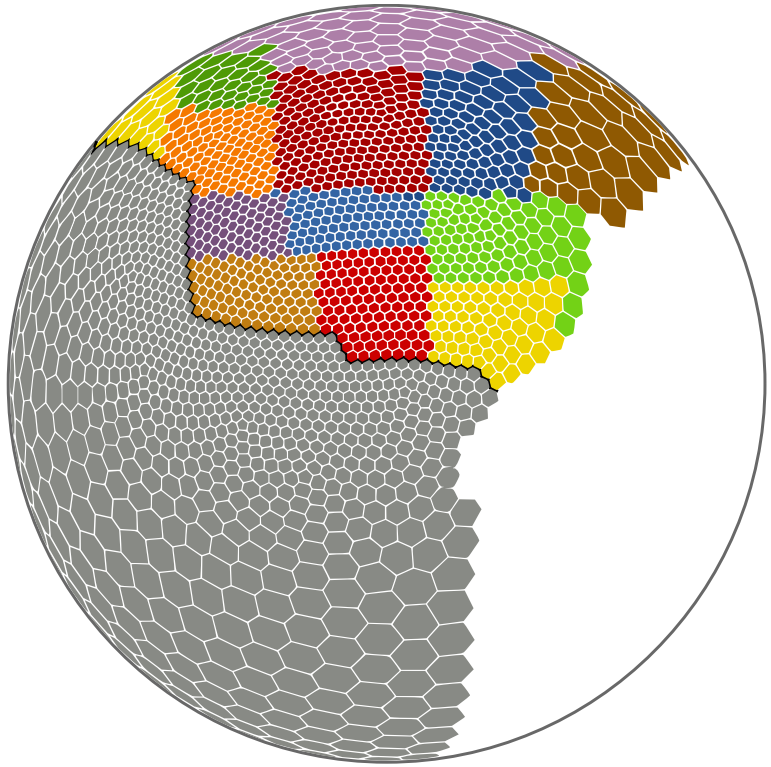
Threading Granularity: Element-Level

- Significant implementation disruption to the code
- Eliminates intra-process halo-exchanges and halo regions
- Computation and communication efficient



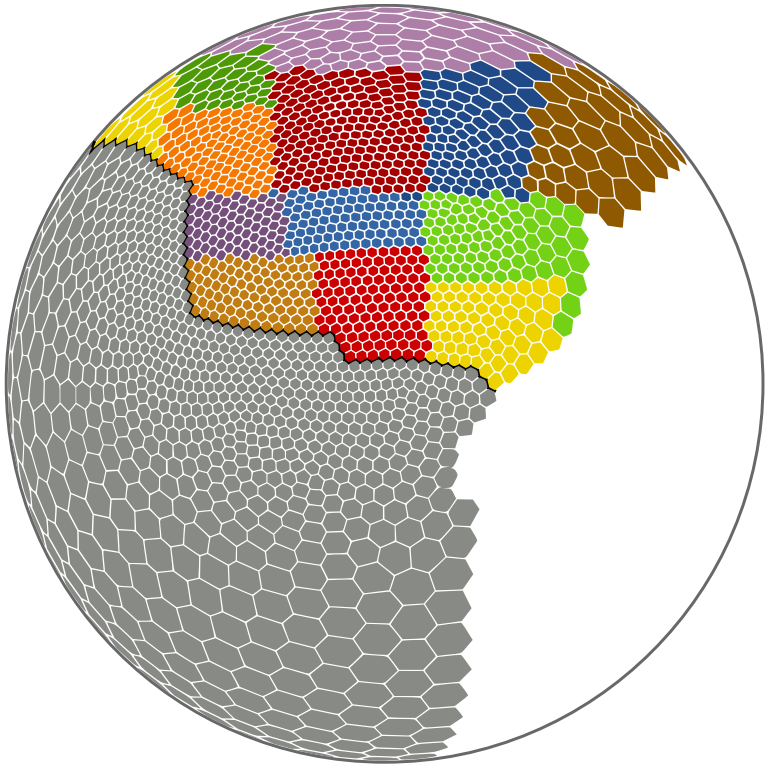
Threading Granularity: Element-Level

- Significant implementation disruption to the code
- Eliminates intra-process halo-exchanges and halo regions
- Computation and communication efficient



Threading Granularity: Element-Level

- Significant implementation disruption to the code
- Eliminates intra-process halo-exchanges and halo regions
- Computation and communication efficient



Element Distributions across Threads

Goal: What is the best approach to distribute mesh elements among threads?

1. Runtime distribution with explicit OpenMP loop directives.
2. Precomputed distributions.
   - Naive/Static.
   - Depth-Aware.

Element Distributions across Threads

Goal: What is the best approach to distribute mesh elements among threads?

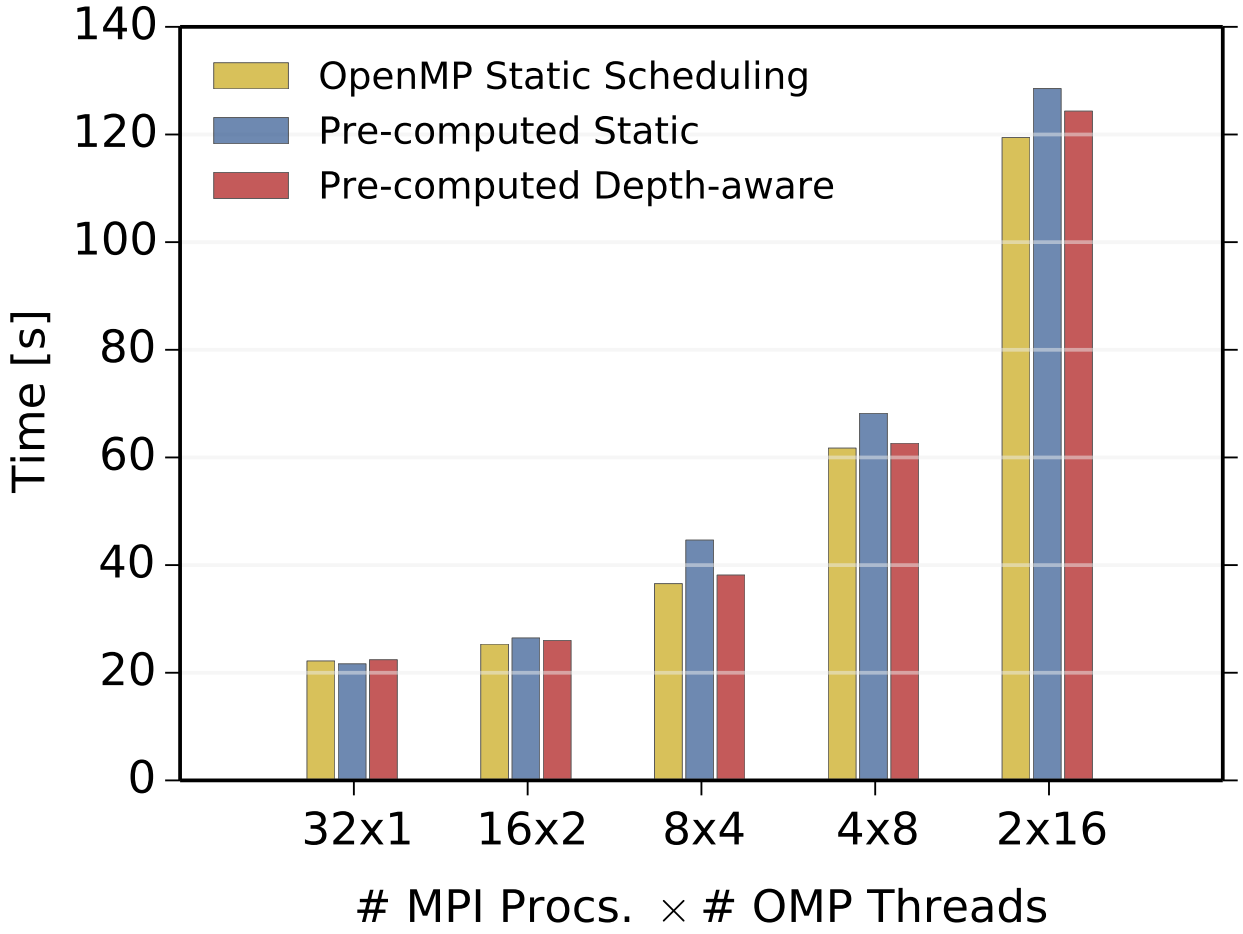
1. Runtime distribution with explicit OpenMP loop directives.
2. Precomputed distributions.
   - Naive/Static.
   - Depth-Aware.

Element Distributions across Threads

Goal: What is the best approach to distribute mesh elements among threads?

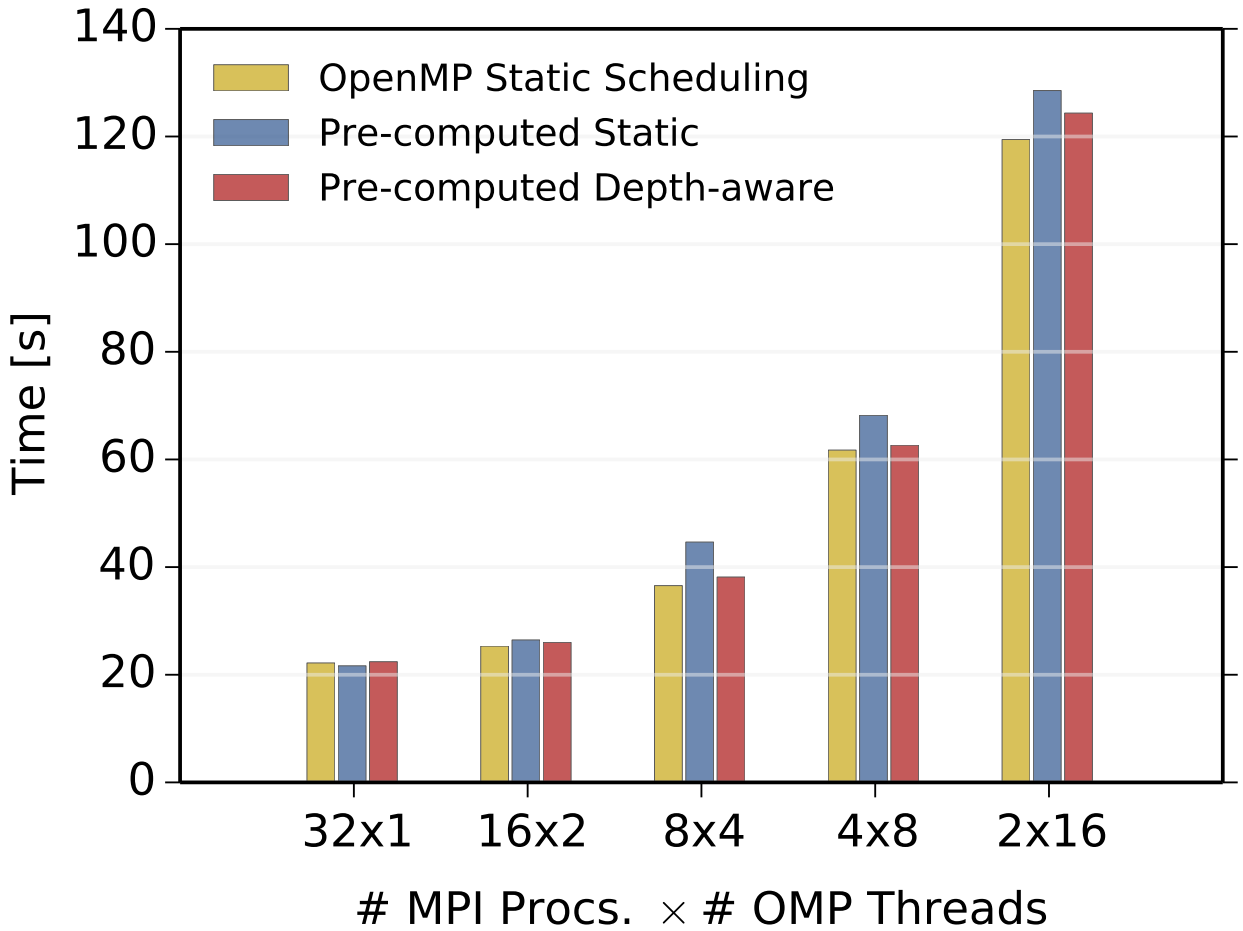
1. Runtime distribution with explicit OpenMP loop directives.
2. Precomputed distributions.
   - Naive/Static.
   - Depth-Aware.

Element Distributions across Threads



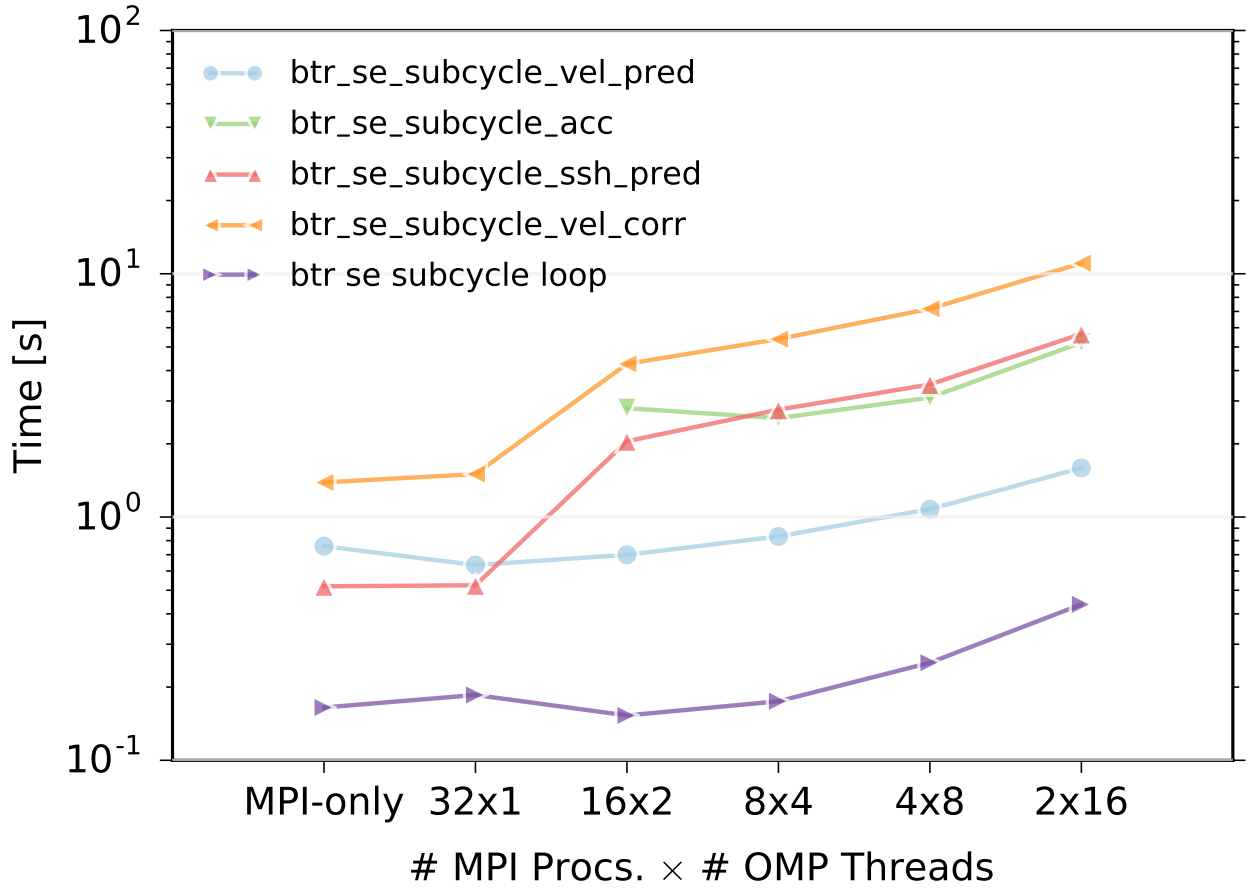
Result: *Explicit OpenMP loop directives.*

Element Distributions across Threads



Result: *Explicit OpenMP loop directives.*

Thread Scaling Bottlenecks

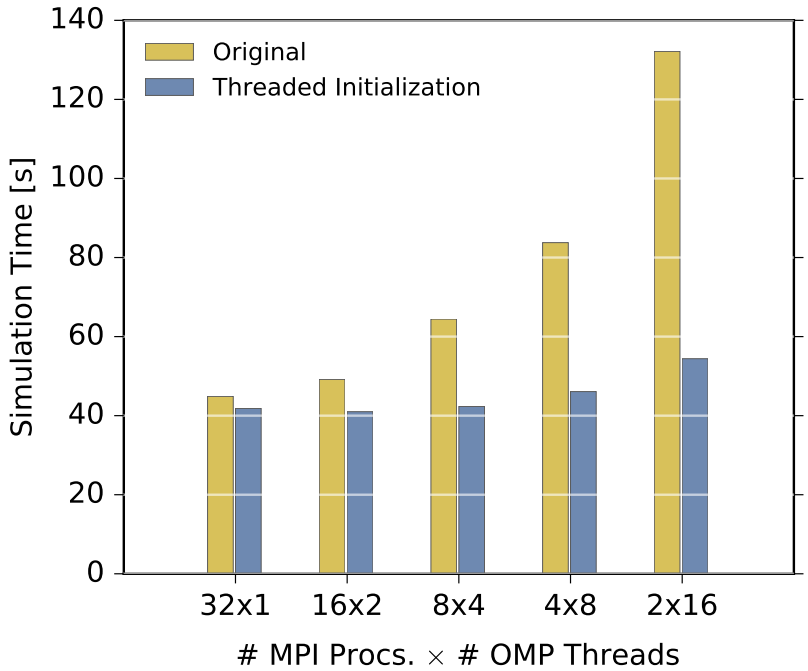


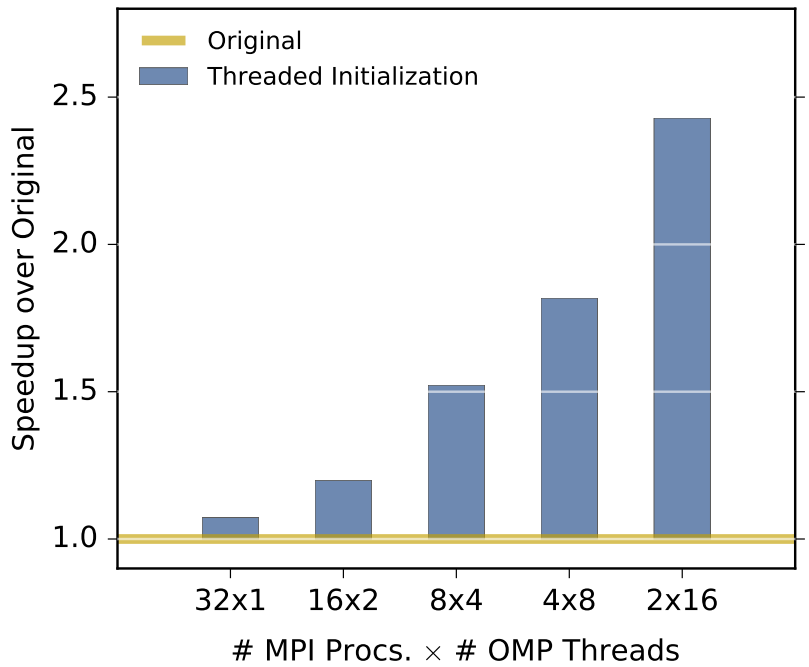
Memory Allocation and Initialization

Goal: Can the Amdahl bottlenecks be minimized?

1. Baseline code: Single thread for memory allocation and initialization.
2. Optimized code: All available threads for initialization.
3. Plus, reordered allocation and initialization events.

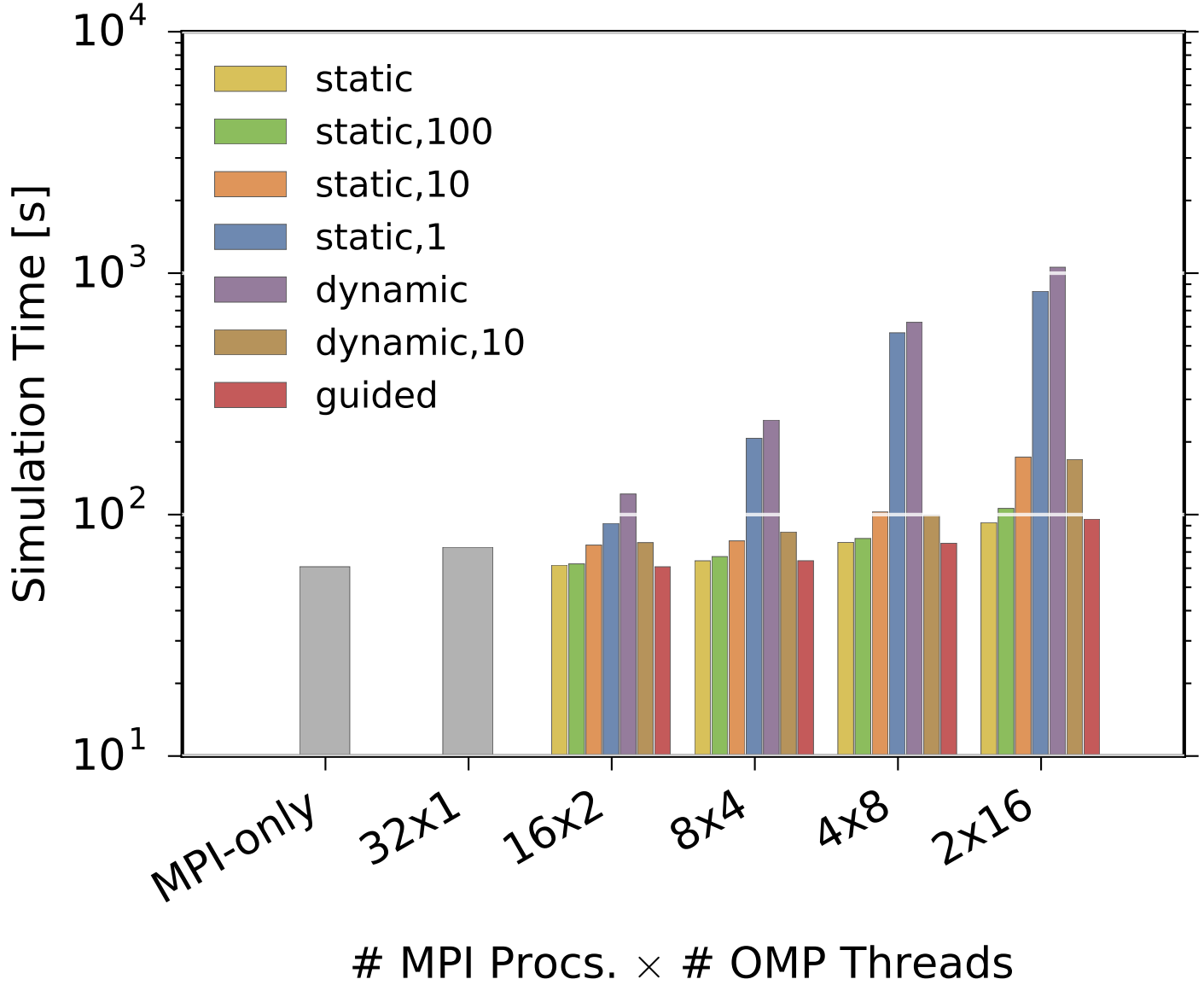
Memory Allocation and Initialization



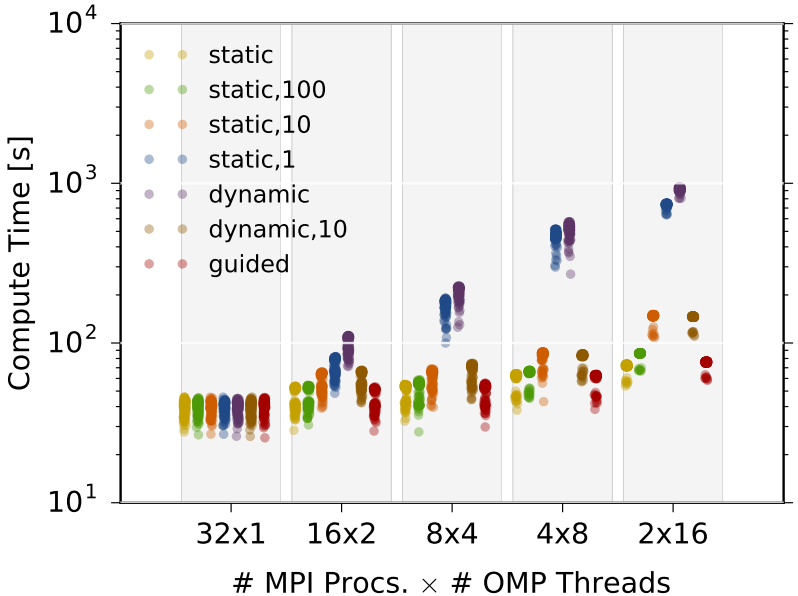


Result: *Multi-threaded memory management.*

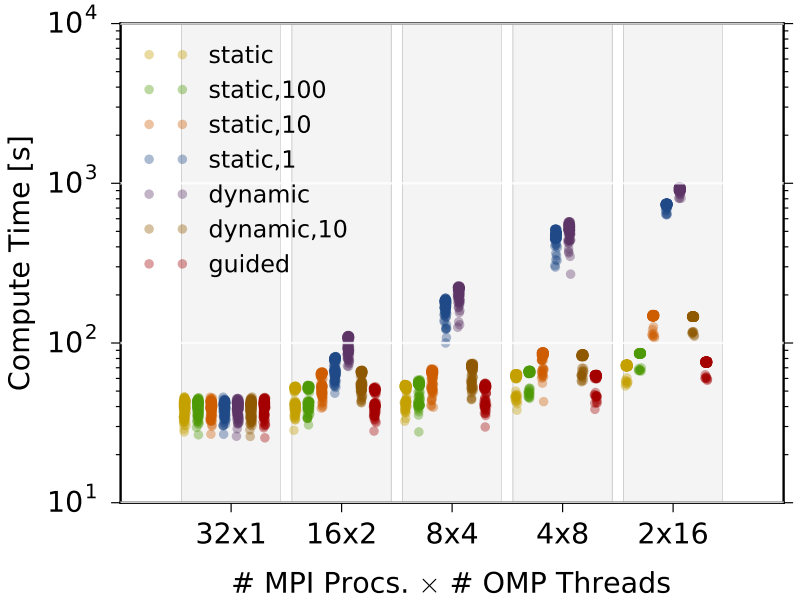
OpenMP Scheduling Policies

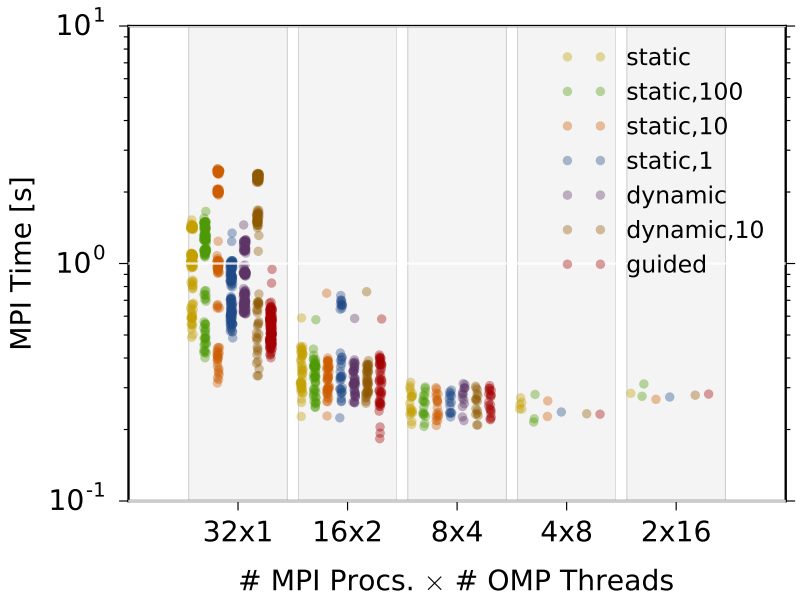


OpenMP Scheduling Policies

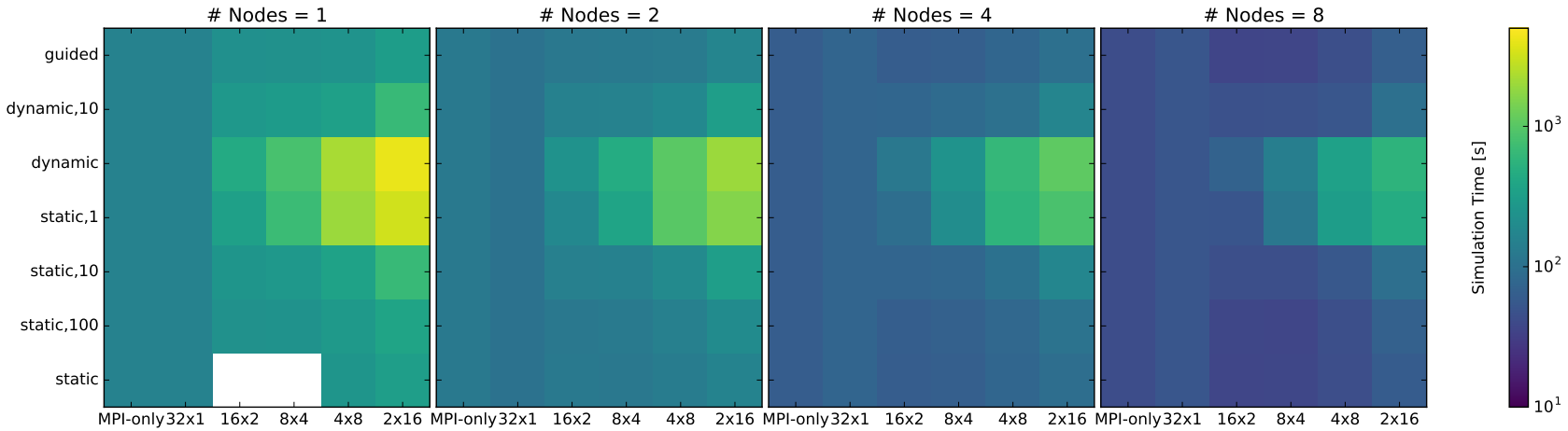


OpenMP Scheduling Policies





OpenMP Scheduling Policies



Result: *Select static or guided scheduling.*

Avoid small chunk sizes.

Balance number of MPI processes and threads: 16×2 *or* 8×4.

Vectorization?

Goal: Can we take advantage of the vector units?

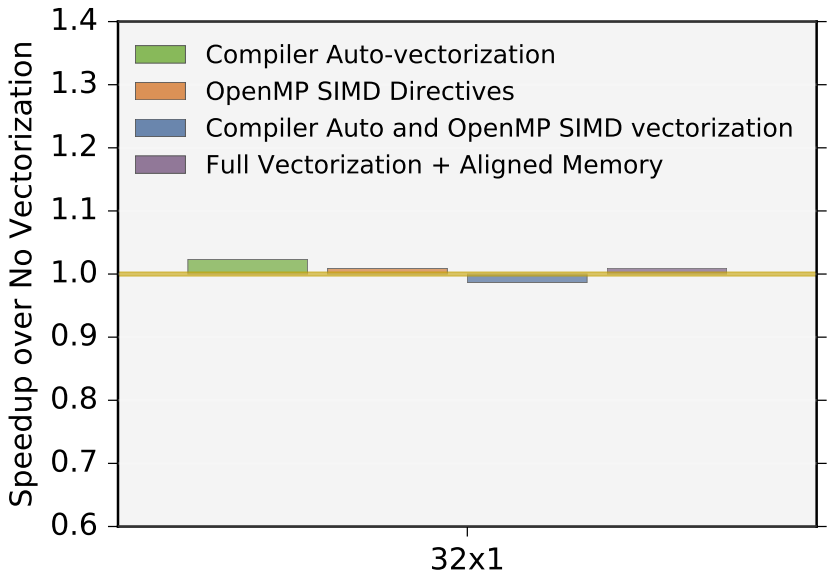
1. None
   `-no-vec, -no-simd`
2. Compiler auto
   `-vec, -simd`
3. OpenMP SIMD only
   `-no-vec, -simd`
4. Full
   `-vec, -simd`
5. Full with aligned memory
   `-align, -vec, -simd`

Result: *Highly memory-bound, making vectorization unnecessary.*

Vectorization?

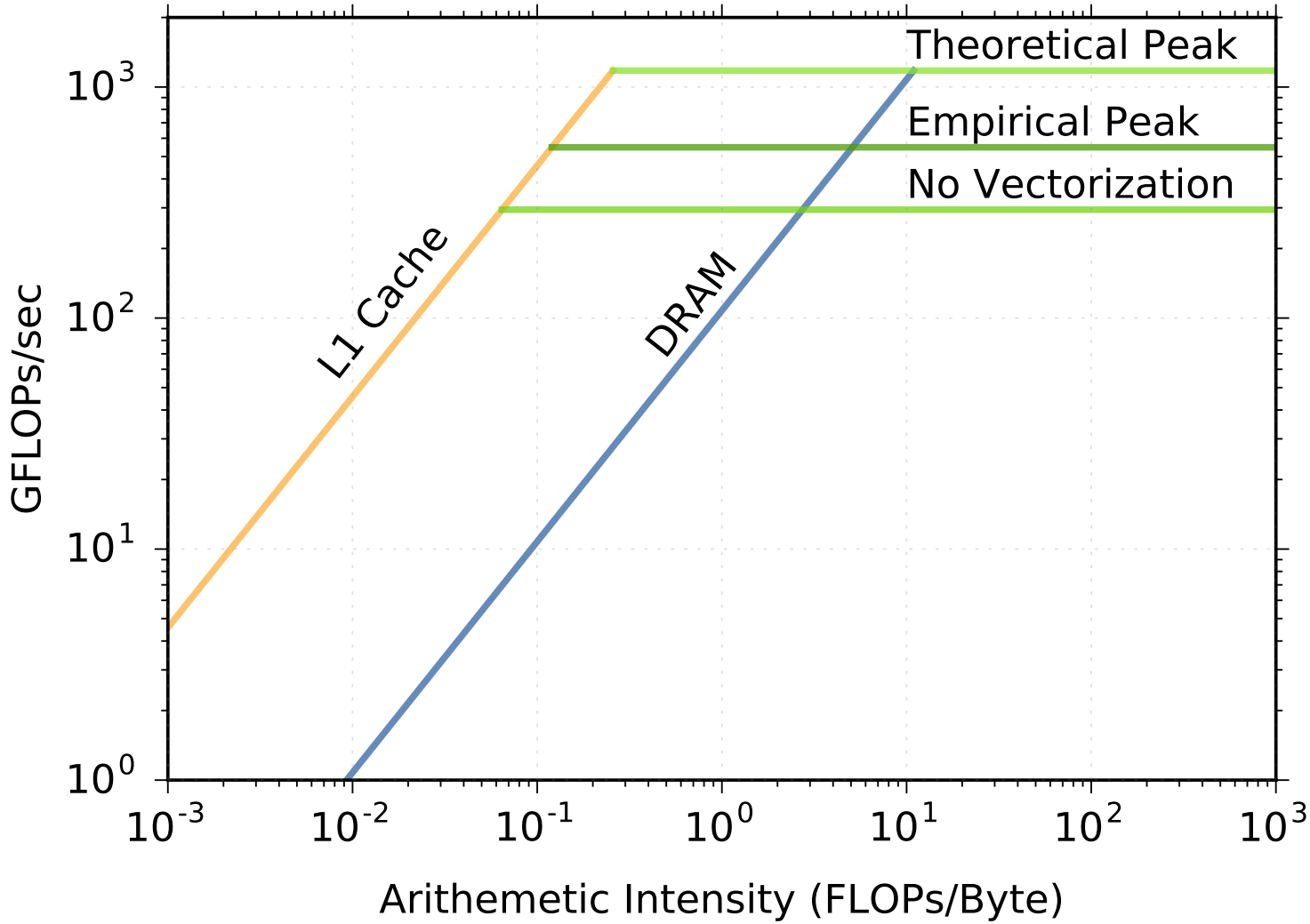
Goal: Can we take advantage of the vector units?

1. None
   `-no-vec, -no-simd`
2. Compiler auto
   `-vec, -simd`
3. OpenMP SIMD only
   `-no-vec, -simd`
4. Full
   `-vec, -simd`
5. Full with aligned memory
   `-align, -vec, -simd`

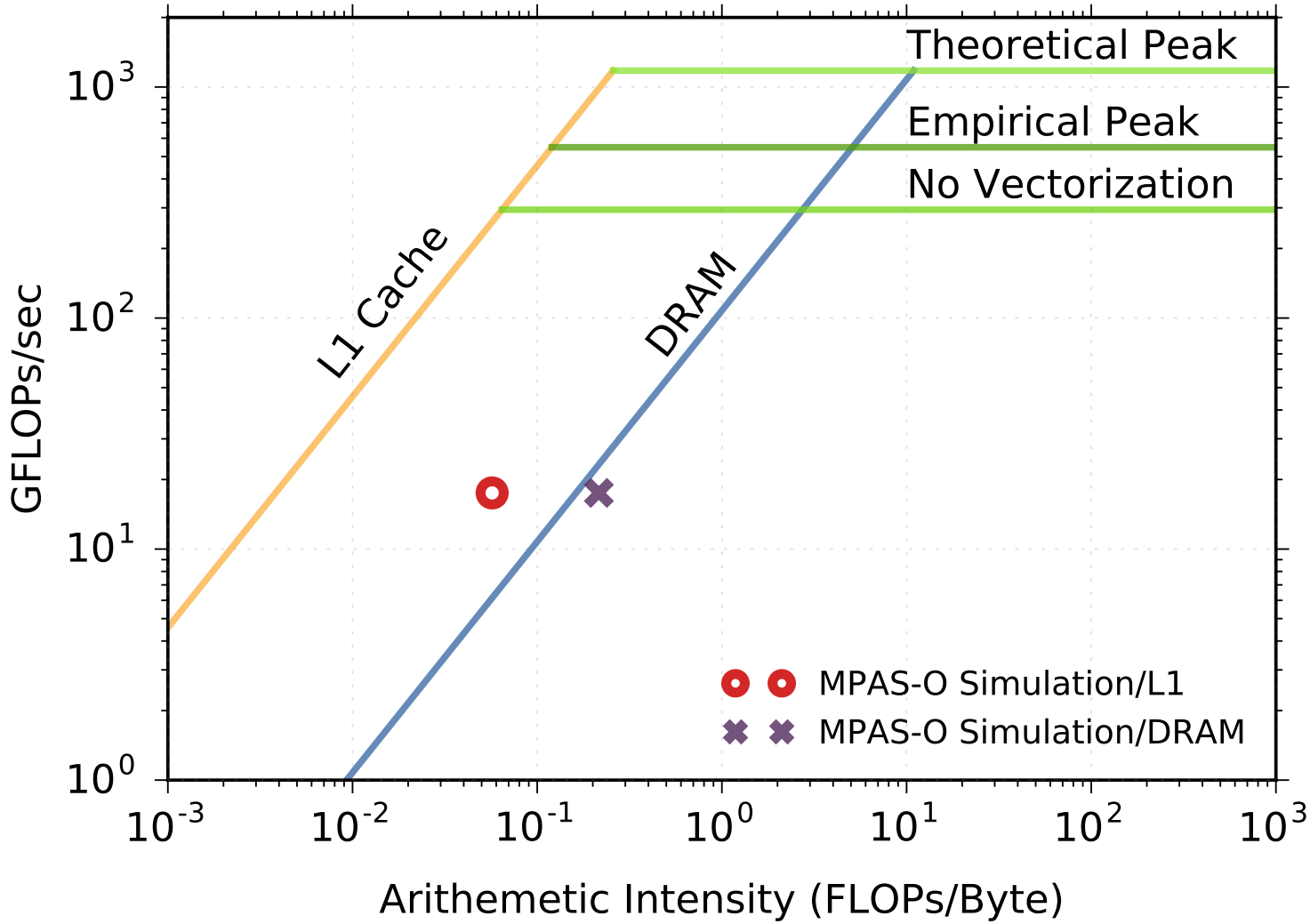


Result: *Highly memory-bound, making vectorization unnecessary.*

Roofline Performance Model

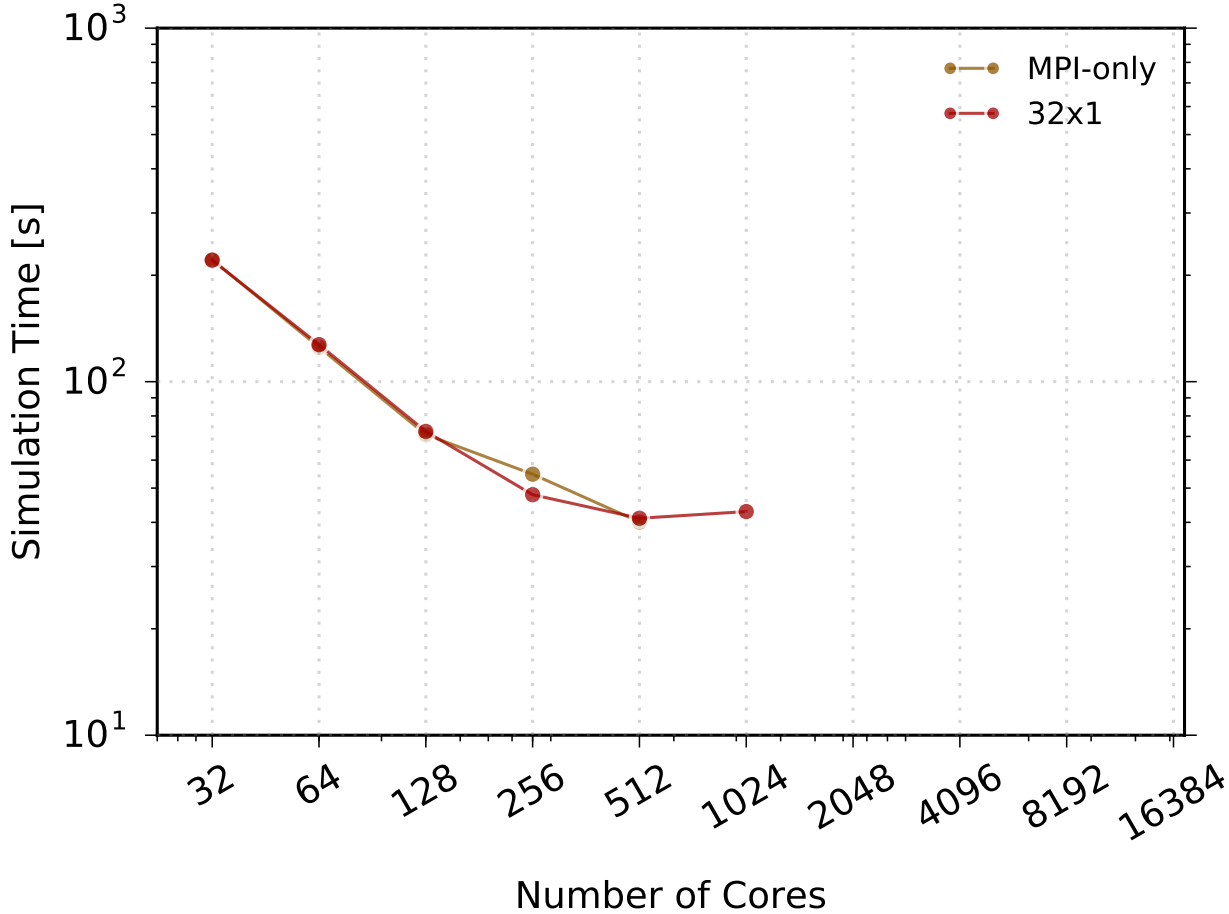


Roofline Performance Model



Strong Scaling: Performance

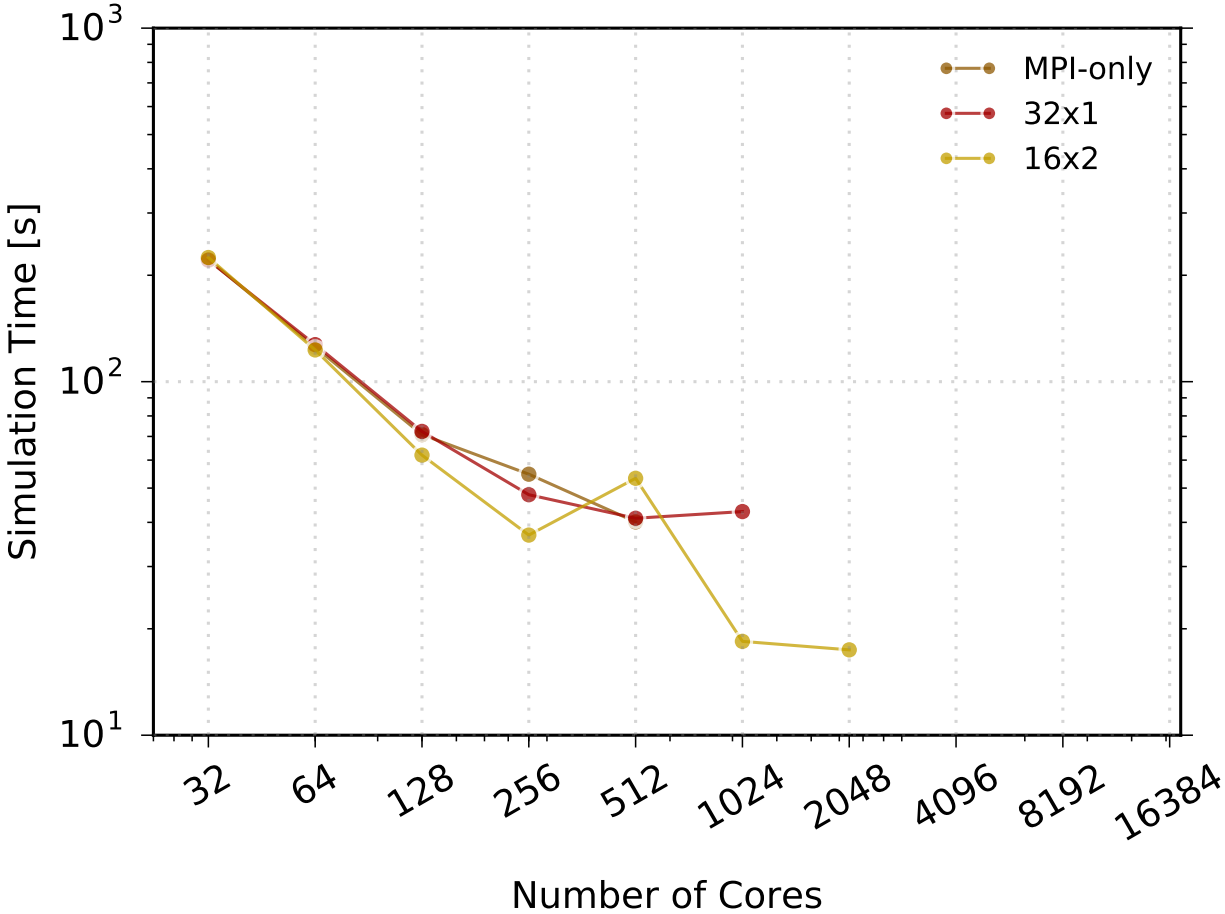
Goal: What is the optimal concurrency?



Result: $3\times$ *speedup for best hybrid* ($n = 64, 8 \times 4$) *w.r.t. best MPI-only* ($n = 16$) *runs.*
$2\times$ *speedup for best hybrid* (8×4) *w.r.t. best MPI-only runs at* $n = 16$.

Strong Scaling: Performance

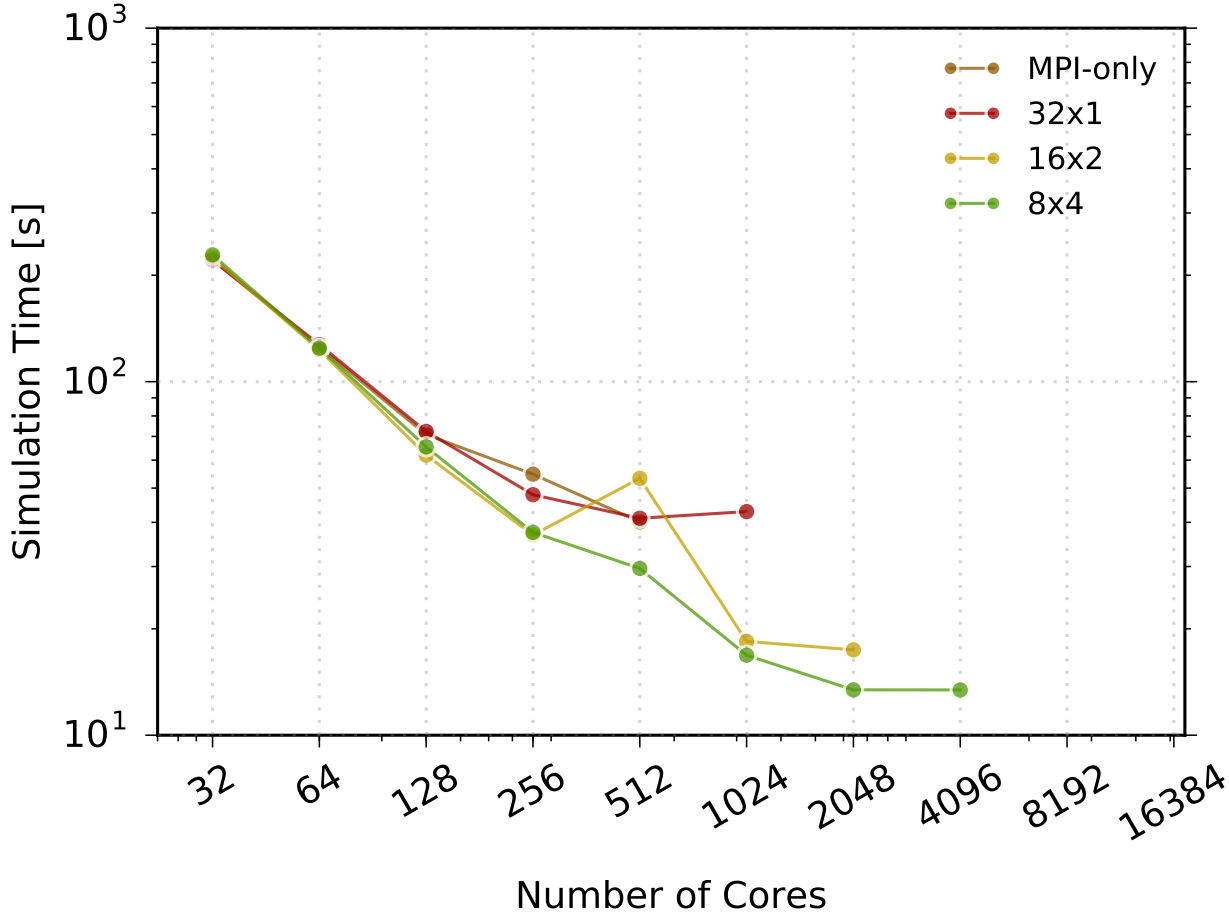
Goal: What is the optimal concurrency?



Result: *3× speedup for best hybrid ($n = 64, 8 \times 4$) w.r.t. best MPI-only ($n = 16$) runs.*
2× speedup for best hybrid (8×4) w.r.t. best MPI-only runs at $n = 16$.

Strong Scaling: Performance

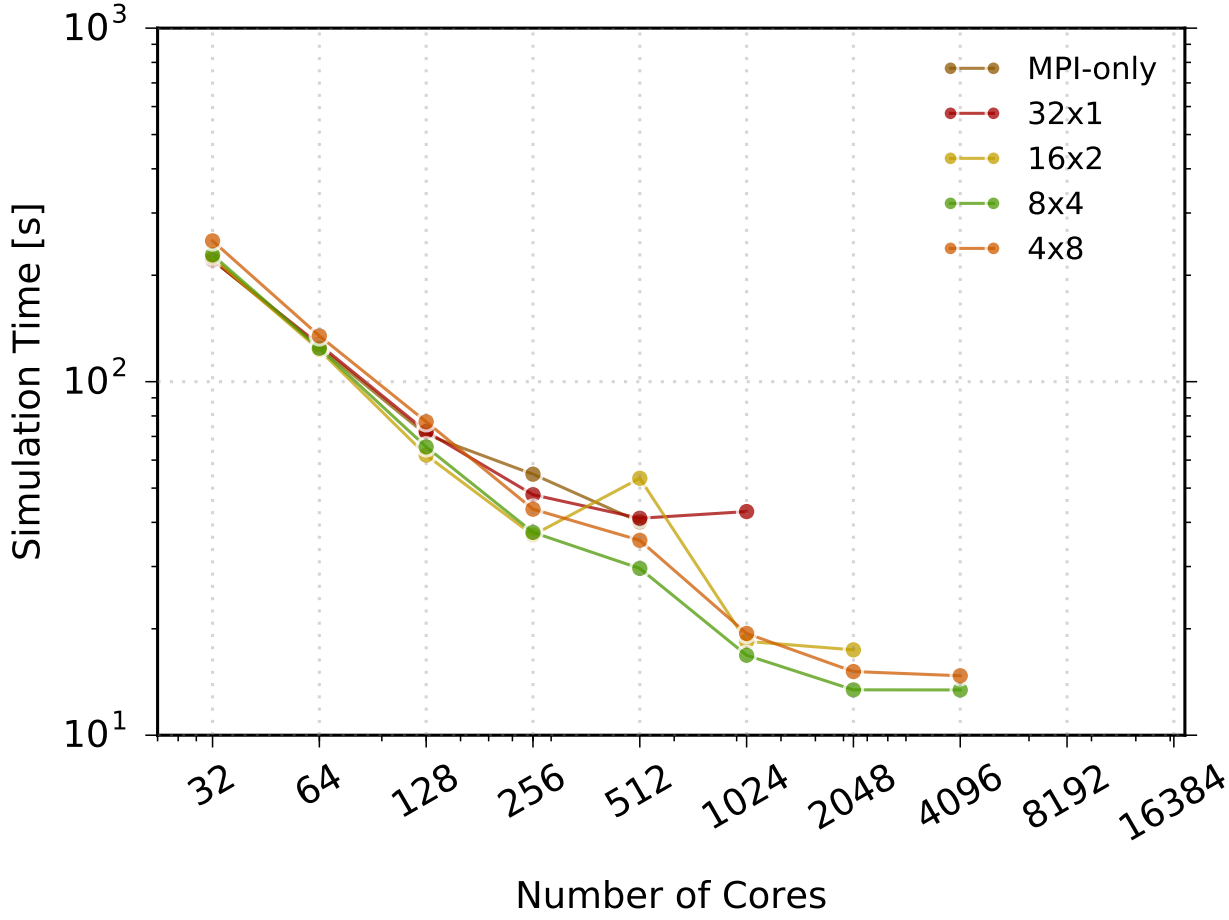
Goal: What is the optimal concurrency?



Result: *3× speedup for best hybrid ($n = 64, 8 \times 4$) w.r.t. best MPI-only ($n = 16$) runs.*
2× speedup for best hybrid (8×4) w.r.t. best MPI-only runs at $n = 16$.

Strong Scaling: Performance

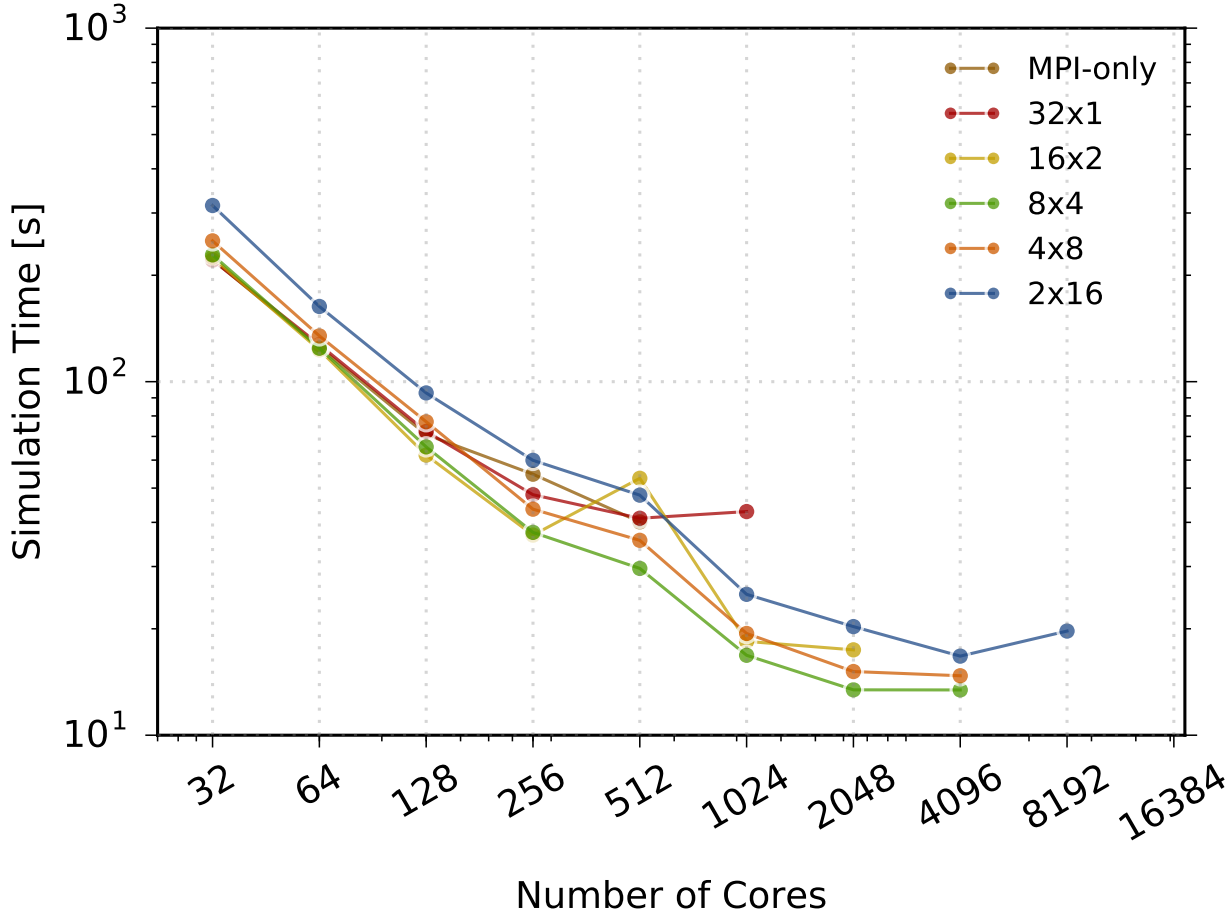
Goal: What is the optimal concurrency?



Result: *3× speedup for best hybrid ($n = 64, 8 \times 4$) w.r.t. best MPI-only ($n = 16$) runs.*
2× speedup for best hybrid (8×4) w.r.t. best MPI-only runs at $n = 16$.

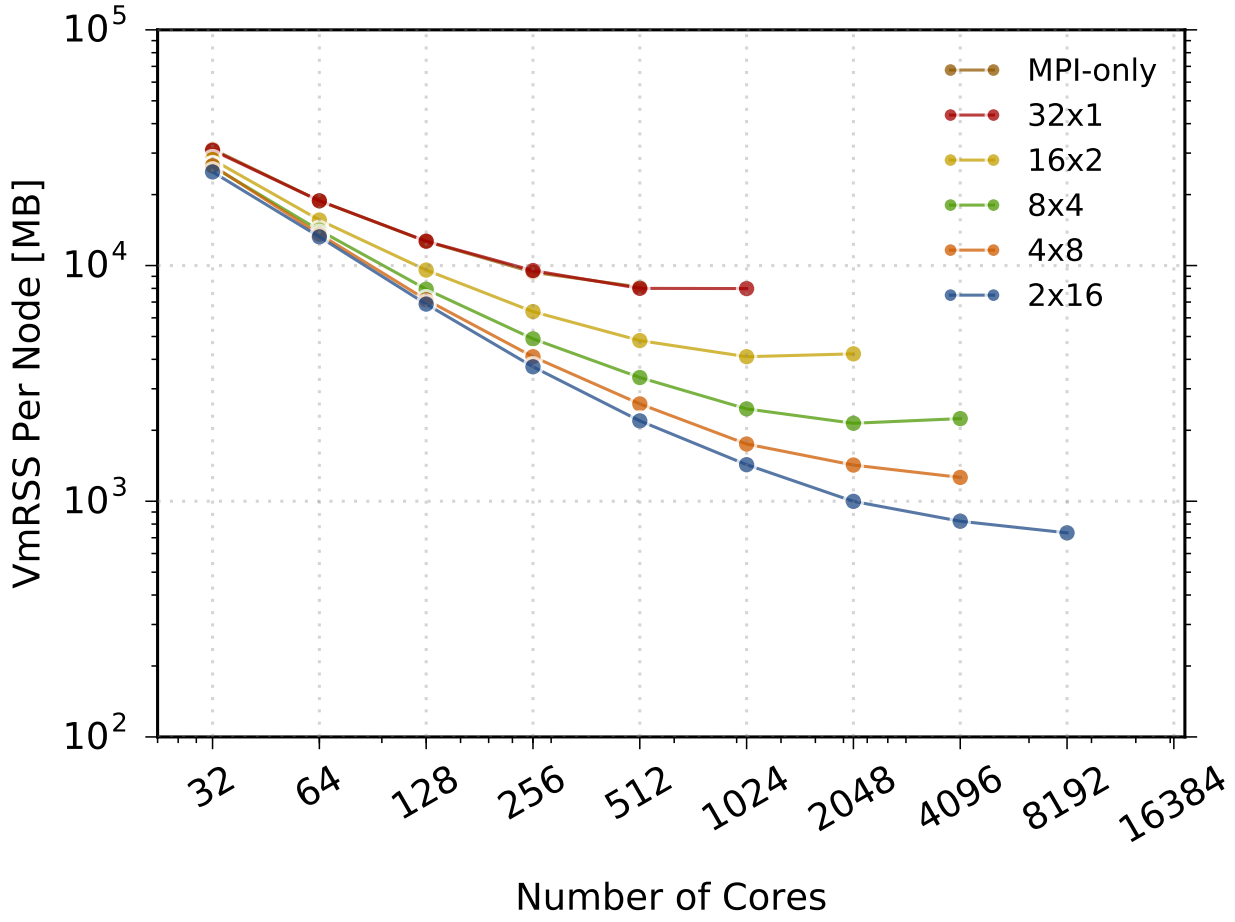
Strong Scaling: Performance

Goal: What is the optimal concurrency?



Result: *3× speedup for best hybrid ($n = 64, 8 \times 4$) w.r.t. best MPI-only ($n = 16$) runs.*
2× speedup for best hybrid (8×4) w.r.t. best MPI-only runs at $n = 16$.

Strong Scaling: Memory Footprint



Minimum memory footprint per process ≈ 250 MB

The working set can possibly completely fit in the HBM on KNL.

End Notes

- Threaded buffer initialization necessary. Observed up to 2.5× speedup on a single node.
- Thread scheduling with larger chunk sizes, such as default static and guided policies, perform best with MPAS-Ocean. Single element chunk size degrades performance by up to an order of magnitude.
- Well balanced number of MPI processes and OpenMP threads (e.g. 8×4) showed best performance.
- Highly irregular memory accesses make depth-awareness redundant.
- Also, being memory bound, vectorization has no benefit.
- Hybrid MPI+OpenMP implementation improves strong scaling at high concurrencies w.r.t. MPI-only implementation. Up to 3× improvement observed for best runs.
- A minimum memory of 250 MB per process needed. Can take advantage of any available HBW memories in upcoming architectures, potentially providing significant performance improvement.

End Notes

- Threaded buffer initialization necessary. Observed up to 2.5× speedup on a single node.
- Thread scheduling with larger chunk sizes, such as default static and guided policies, perform best with MPAS-Ocean. Single element chunk size degrades performance by up to an order of magnitude.
- Well balanced number of MPI processes and OpenMP threads (e.g. 8×4) showed best performance.
- Highly irregular memory accesses make depth-awareness redundant.
- Also, being memory bound, vectorization has no benefit.
- Hybrid MPI+OpenMP implementation improves strong scaling at high concurrencies w.r.t. MPI-only implementation. Up to 3× improvement observed for best runs.
- A minimum memory of 250 MB per process needed. Can take advantage of any available HBW memories in upcoming architectures, potentially providing significant performance improvement.

End Notes

- Threaded buffer initialization necessary. Observed up to 2.5× speedup on a single node.
- Thread scheduling with larger chunk sizes, such as default static and guided policies, perform best with MPAS-Ocean. Single element chunk size degrades performance by up to an order of magnitude.
- Well balanced number of MPI processes and OpenMP threads (e.g. 8×4) showed best performance.
- Highly irregular memory accesses make depth-awareness redundant.
- Also, being memory bound, vectorization has no benefit.
- Hybrid MPI+OpenMP implementation improves strong scaling at high concurrencies w.r.t. MPI-only implementation. Up to 3× improvement observed for best runs.
- A minimum memory of 250 MB per process needed. Can take advantage of any available HBW memories in upcoming architectures, potentially providing significant performance improvement.

End Notes

- Threaded buffer initialization necessary. Observed up to 2.5× speedup on a single node.
- Thread scheduling with larger chunk sizes, such as default static and guided policies, perform best with MPAS-Ocean. Single element chunk size degrades performance by up to an order of magnitude.
- Well balanced number of MPI processes and OpenMP threads (e.g. 8×4) showed best performance.
- Highly irregular memory accesses make depth-awareness redundant.
- Also, being memory bound, vectorization has no benefit.
- Hybrid MPI+OpenMP implementation improves strong scaling at high concurrencies w.r.t. MPI-only implementation. Up to 3× improvement observed for best runs.
- A minimum memory of 250 MB per process needed. Can take advantage of any available HBW memories in upcoming architectures, potentially providing significant performance improvement.

End Notes

- Threaded buffer initialization necessary. Observed up to 2.5× speedup on a single node.
- Thread scheduling with larger chunk sizes, such as default static and guided policies, perform best with MPAS-Ocean. Single element chunk size degrades performance by up to an order of magnitude.
- Well balanced number of MPI processes and OpenMP threads (e.g. 8×4) showed best performance.
- Highly irregular memory accesses make depth-awareness redundant.
- Also, being memory bound, vectorization has no benefit.
- Hybrid MPI+OpenMP implementation improves strong scaling at high concurrencies w.r.t. MPI-only implementation. Up to 3× improvement observed for best runs.
- A minimum memory of 250 MB per process needed. Can take advantage of any available HBW memories in upcoming architectures, potentially providing significant performance improvement.

End Notes

- Threaded buffer initialization necessary. Observed up to 2.5× speedup on a single node.
- Thread scheduling with larger chunk sizes, such as default static and guided policies, perform best with MPAS-Ocean. Single element chunk size degrades performance by up to an order of magnitude.
- Well balanced number of MPI processes and OpenMP threads (e.g. 8×4) showed best performance.
- Highly irregular memory accesses make depth-awareness redundant.
- Also, being memory bound, vectorization has no benefit.
- Hybrid MPI+OpenMP implementation improves strong scaling at high concurrencies w.r.t. MPI-only implementation. Up to 3× improvement observed for best runs.
- A minimum memory of 250 MB per process needed. Can take advantage of any available HBW memories in upcoming architectures, potentially providing significant performance improvement.

End Notes

- Threaded buffer initialization necessary. Observed up to 2.5× speedup on a single node.
- Thread scheduling with larger chunk sizes, such as default static and guided policies, perform best with MPAS-Ocean. Single element chunk size degrades performance by up to an order of magnitude.
- Well balanced number of MPI processes and OpenMP threads (e.g. 8×4) showed best performance.
- Highly irregular memory accesses make depth-awareness redundant.
- Also, being memory bound, vectorization has no benefit.
- Hybrid MPI+OpenMP implementation improves strong scaling at high concurrencies w.r.t. MPI-only implementation. Up to 3× improvement observed for best runs.
- A minimum memory of 250 MB per process needed. Can take advantage of any available HBW memories in upcoming architectures, potentially providing significant performance improvement.

Acknowledgements

- Authors from LBNL were supported by U.S. Department of Energy Office of Science's Advanced Scientific Computing Research program under contract number DE-AC02-05CH11231.
- Authors from LANL were supported by U.S. Department of Energy Office of Science's Biological and Environmental Research program.
- Resources at NERSC were used, supported by DOE Office of Science under Contract No. DE-AC02-05CH11231.

Thank you!