Exploiting Thread Parallelism for Ocean Modeling on Cray XC Supercomputers

Abhinav Sarje*, Douglas W. Jacobsen†, Samuel W. Williams*, Todd Ringler†, Leonid Oliker*
*Lawrence Berkeley National Laboratory {asarje,swwilliams,loliker}@lbl.gov
†Los Alamos National Laboratory {douglasj,ringler}@lanl.gov

***Abstract*—The incorporation of increasing core counts in modern processors used to build state-of-the-art supercomputers is driving application development towards exploitation of thread parallelism, in addition to distributed memory parallelism, with the goal of delivering efficient high-performance codes. In this work we describe the exploitation of threading and our experiences with it with respect to a real-world ocean modeling application code, MPAS-Ocean. We present detailed performance analysis and comparisons of various approaches and configurations for threading on the Cray XC series supercomputers.**

I. Introduction

Global climate modeling is considered a grand challenge problem due to the spatial and temporal scales required to accurately simulate the involved phenomena. Temporal scales for climate models are on the order of centuries, while spatial scales have resolutions up to several kilometers. The Model for Prediction Across Scales (MPAS) is an Earth-system modeling framework collaboratively developed at the Los Alamos National Laboratory and the National Center for Atmospheric Research. The MPAS framework has four dynamical cores – shallow water, atmosphere, land ice, and ocean. In this paper, we focus on the ocean core, MPAS-Ocean [1], which has been primarily implemented in Fortran.

Two-dimensional spherical centroidal Voronoi tessellations (SCVT) form the underlying discretized mesh for MPAS-Ocean, covering Earth's oceans [1], [2], [3]. These meshes are unstructured and support variable resolutions across the spatial domain. An additional third dimension also exists as vertical columns for each mesh element, and represents the ocean depth. Data structures are built on top of the mesh where, although the horizontal structure is staggered, the vertical structure, representing ocean depth, is regular. The requirements for climate studies need simulations on a large number of horizontal degrees of freedom, $O(10^6)$ to $O(10^7)$ with about $O(10^2)$ vertical levels per horizontal degree of freedom, and demand performance many orders of magnitude faster than real-time. Combined, these necessitate efficient execution on highly parallel supercomputing systems to drive useful analysis. In this paper, we describe the utilization of on-node threading to effectively exploit the high degrees of hierarchical parallelism available in the state-of-the-art systems, with a focus on two Cray XC series supercomputers.

Achieving high parallel efficiency on applications like MPAS-Ocean, which use unstructured meshes, can be challenging due to the various factors characteristic of such meshes affecting the overall performance, including imbalance across processes and threads, irregular data layouts and pointer jumping, inefficient communication patterns involving boundary data exchanges, and non-uniform amounts of work at each mesh element due to varying number of valid vertical levels (spatially varying ocean depth). Additionally, this application utilizes deep halo regions, where several layers of halo elements are needed, aggravating the impact of imbalance, as illustrated in Fig. 1. Furthermore, for simplicity and ease of implementation, MPAS-Ocean uses padded mesh data structures where each mesh element has memory allocated for the overall maximum vertical depth, and the redundant levels at each element are marked as invalid, as shown in Fig. 2.

Improving performance of this application requires addressing all these challenges, including better data organization to improve cache efficiency and intelligent mesh partitioning that takes the halo regions into account to reduce imbalances in both the work load per process as well as communication volumes [4]. Exploitation of on-node shared-memory parallelism through threading can also enhance the performance due to factors such as reduced memory usage by the elimination of deep inter-core halo regions, lightweight concurrency management within NUMA domains, reduced network message contention, and larger (more efficient) MPI messaging. Threading also can benefit from data sharing among threads of a process through better cache data re-use. In this paper, we explore and demonstrate the utilization of thread parallelism in the MPAS Ocean core. The primary contributions of this work are:

- Implementation of threading into the MPAS ocean core along with performance optimizations.
- Detailed analysis of various threading configurations and scheduling to guide the selection of options delivering best performance.
- Analysis of application scaling, with respect to both execution time and energy consumption, to identify performance optimal configurations and concurrency levels for a given mesh size.
- Additionally, we explore the potential performance enhancement in forthcoming CPU architectures, such as the Intel Knights Landing processors, through the utilization of new architectural features such as increased thread parallelism and high-bandwidth memory.

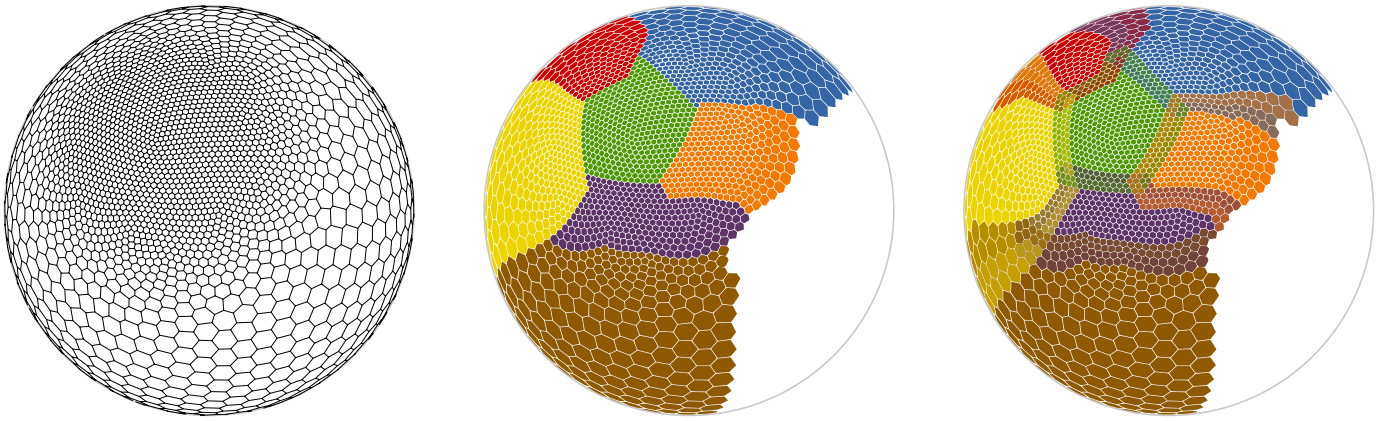

Fig. 1. (Left) An example of a variable resolution mesh covering Earth's surface. (Center) A mesh partitioning into blocks, each represented by a different color, covering only the oceans (excluding land). (Right) Deep halo regions with two layers of halo cells are shown in shades of color of the corresponding blocks. In the strong scaling limit, these halos begin to dominate the local storage requirements.

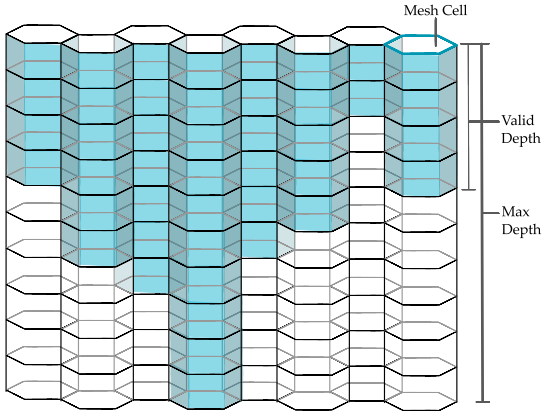

Fig. 2. Varying ocean depth across the mesh elements with padding with unused levels are shown.

II. Exposing Parallelism for Threading

MPAS-Ocean meshes are composed of *mesh elements*, namely, cells, vertices and edges. These meshes are decomposed into partitions called *blocks* through the use of mesh partitioners [4], each with a unique subset of elements, which are ideally contiguous. These blocks are distributed across the compute processors via MPI, with each MPI process owning one or more blocks. Deep halo regions, typically with depth of three, used for communicating data across these blocks, are constructed for each block to provide data continuity. With such decomposition flexibility, two primary levels of on-node parallelism are exposed: *block-level* and *element-level*. By assigning multiple blocks per process, threading can be exploited via parallelization over the list of blocks owned by a process. Hence, each process would ideally own a number of blocks that is equal to or a multiple of the number of available threads to ensure a well-balanced computation. This level provides coarse-grained threading, and results in a minimal implementation disruption to the code. Unlike the halo-exchanges across different compute nodes requiring off-node data movements, simple intra-node halo-exchanges can be performed among the blocks owned by a single process without the need of any explicit communication.

Given the current trend of rapidly increasing core counts on modern multicore processors, threading over blocks may yield insufficient parallelism, leading to under-utilization of the available compute resources. Moreover, the deep intra-block (i.e. inter-thread) halo regions persist and consume memory, which may lead to any communication performance gain being outweighed by this overhead. An alternate to the threading implementation involves assigning a single or small number of blocks per MPI process, while threading over the mesh elements (i.e. cells, edges and vertices) contained in these blocks. This approach eliminates intra-process halo exchanges as well as all inter-thread deep halo regions. As such, it is both communication- and memory-efficient, while providing a finer-grained parallelism. On the other hand, this approach requires greater code implementation disruption and efforts. Both of these threading approaches are shown in Fig. 3. Ultimately, our performance analysis showed higher performance with element-level threading compared with the block-level approach, and we therefore select this decomposition for our study.

In order to implement threading within the ocean core, we took an incremental approach that preserves not only the ability to run flat MPI, but also allows users to tune performance for the optimal balance of processes and threads. Additionally, it ensures that the ocean core threading strategy is kept independent from other cores in the MPAS system, which may or may not implement threading. To that end, we embrace a single program multiple data (SPMD) approach to OpenMP parallelism via a global `ompparallel` region enclosing the ocean core simulation timestepping, where all threads execute all operations in a timestep concurrently. This approach was motivated by the the assumption that frequent creation of parallel regions is an impediment to strong scaling. Since some operations on shared memory must be executed sequentially, such as allocations and deallocations of large data buffers, a combination of `omp_get_thread_num` and

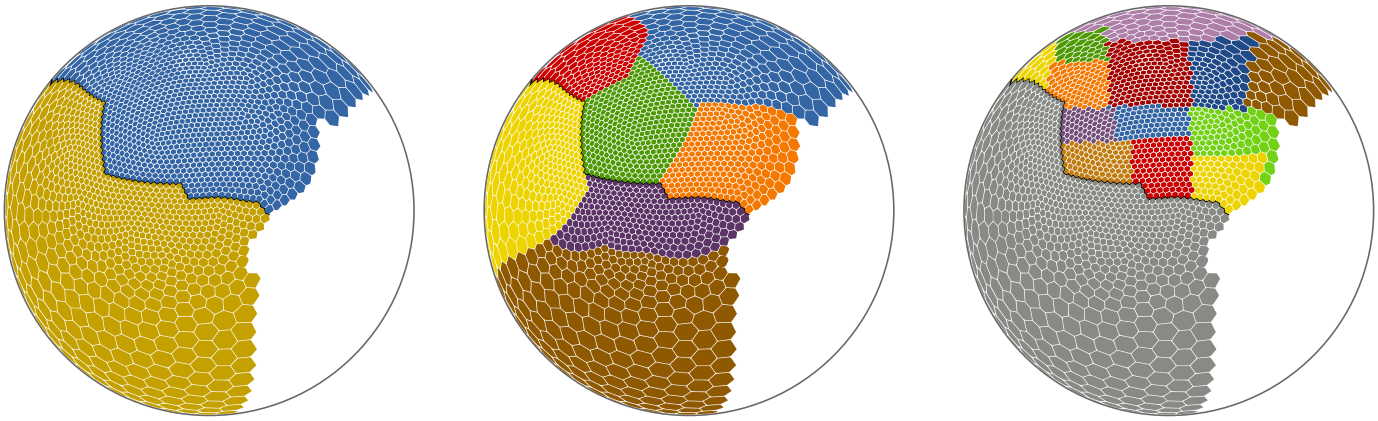

Fig. 3. (Left) Cell assignments to two processes, each shown in different color. (Center) Block-level threading where each set of cells assigned to a process is further decomposed into blocks, each assigned to a different thread, shown in different colors. (Right) Element-level threading where cells assigned to a process are distributed among all threads depending on the scheduling policy used, the cells assigned to a thread shown in different colors for one of the processes.

`ompbarrier` ensures that exactly one thread calls the memory management routines to allocate and deallocate the buffers required within each routine, and no thread can access the data before it is allocated. Similar constructs are used for calls to the MPI communication (halo-exchanges) and I/O routines where a single thread per process is responsible for performing these operations. All computational loops over the mesh elements in the ocean core are threaded with a `!$omp do schedule(runtime)` in order to enable tuning of the thread scheduling algorithm and chunk size used at runtime. However, computations over the vertical depth are not threaded, instead we rely on the vectorization and instruction-level parallelism of the underlying processor to maximize performance over these relatively short loops.

It should be noted that although computation on each vertical column in the mesh is limited by the actual depth of the ocean at that point on the Earth, the underlying data structure is two-dimensional with a common maximum depth across all columns throughout the mesh, as illustrated in Fig 2. The levels beyond the actual depth at a cell are invalid and not involved in computations. Although this greatly simplifies indexing and implementation, micro-benchmarks suggest that this approach can impede efficient memory access and memory requirements, and thus negatively impact both inter-process and inter-thread load balancing. The exploration of remedies of these impediments is out of scope of this paper and we leave it as a future work.

III. Computational Environment and Meshes Used

The current and near-future state-of-the-art supercomputers utilize multi- and many-core processors as the primary workhorse. With the trend of increasing number of cores available on a compute node, the need to effectively thread codes has become more important now than ever in order to efficiently utilize the computational power offered by these architectures. In this work we focus on the current Cray XC series supercomputers installed at the National Energy Research and Scientific Computing Center (NERSC), and present the performance on the following system:

- Cray XC40 (*Cori Phase 1*): This system consists of 1,630 dual-socket compute nodes housing 16-core Intel Haswell processors, providing a total of 32 cores and 128GB memory per node with two NUMA domains, and connected through the Cray Aries interconnect. This is phase 1 of the Cori system, which will ultimately house the Intel Knights Landing processors.

We use Intel compilers version 16.0 and collect performance data using the TAU performance tool [5] in conjunction with hardware counters through PAPI [6]. Additional measurements of number of FLOPs and memory bandwidth are performed using Intel SDE and Intel Vtune tools.

In the experiments presented in this paper, the simulations are performed using two different input meshes. These meshes have been selected based on their sizes (i.e. horizontal resolutions and vertical depths) and the goal of the experiments. The details of these two meshes are as follows:

1) Mesh 1: This mesh represents discretization of the Earth's oceans at a uniform resolution of 60 kms. It consists of a total of 114,539 cells representing the mesh surface, with ocean depths represented by a maximum of 40 vertical levels.
2) Mesh 2: This is a variable resolution mesh with discretization ranging from 30 kms at the North Atlantic region to 60 kms elsewhere, resulting in a total of 234,095 horizontal cells. Each cell has a maximum of 60 vertical levels representing the ocean depths.

IV. Threading Implementation

The common models to utilize multiple on-node compute cores include distributed-memory (e.g. MPI), shared-memory (e.g. threading), or a hybrid of the two. Although each model has its advantages, the overall performance benefits are generally problem dependent. MPAS-Ocean was originally developed using only MPI parallelism. In this section, we describe

our hybrid implementation, by incorporating threading using OpenMP. As previously discussed in Section II, we select threading at the mesh element level (cells, edges and vertices). In this application, the use of such on-node threading across multiple cores has a number of potential benefits, including:

- Executing threads instead of multiple MPI processes across cores now demands less aggregate memory and reduces communication traffic.
- Since threads share the data corresponding to the blocks assigned to their parent process, the total number of blocks required is effectively reduced by a factor of the number of threads, thereby reducing the size of the halo regions. This contributes to lowered overall communication volume, as well as reduced computations needed on the halo elements.
- MPAS-Ocean can now scale to higher concurrencies compared to the MPI-only version, primarily due to reduced resource requirements.
- Caches now have improved utilization through better data reuse across the threads.

With increasing depth of memory hierarchies in modern processor architectures, reduction in the memory footprint as a result of threading is expected to have further increase in performance. This enhancement is particularly significant when the working data set is able to fit into the lower-latency high-bandwidth memory levels closer to the compute cores, minimizing the amount of data fetching required from the higher-latency low-bandwidth memories, while also helping hiding the latency overheads.

A. Element distributions

In our work, we examine several different approaches to distribution of mesh elements across threads. A straightforward approach to implement threading using OpenMP is to use the loop directives '!$omp do' with each of the element loops explicitly. Another approach is to use a pre-computed distribution of elements among the threads to specify the range of elements a given thread is responsible for computing. This latter approach gives a finer control over the distribution of elements across the threads, compared to the predefined scheduling policies with explicit OpenMP directives, and allows for incorporating any prior knowledge about the elements in computing the distributions. Apart from a naive static division of elements across the threads, equivalent to the OpenMP static scheduling, we also implemented a *depth-aware* distribution, where instead of equally distributing the number of elements across threads, the total work is equally distributed in the anticipation of achieving a better compute load balance among the threads due to the varying valid depths. The former approach requires the effort of going over all the loops over elements throughout the ocean core code and adding the OpenMP directives to each of them. In contrast, the latter approach simply requires calculating 'begin' and 'end' indices for each thread corresponding to each of the element types during application initialization and using these indices in the element loops. Implementing a different distribution only requires updating this calculation, leaving rest of the code untouched.

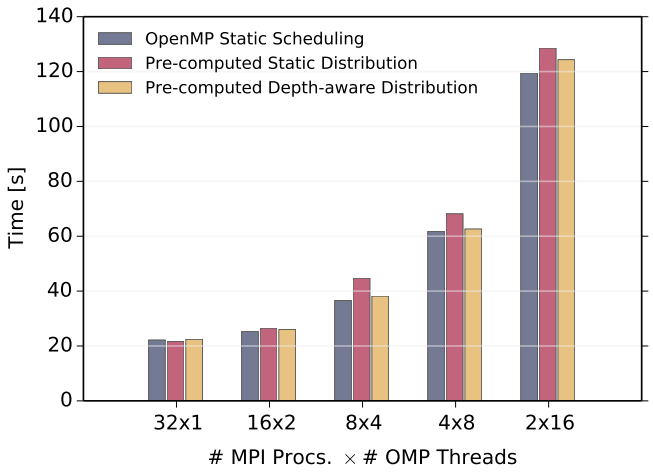

Fig. 4. Performance comparison with different element distribution approaches: The performance as execution times for the routine `advection` of MPAS-Ocean is shown with the approaches considered to implement threading in MPAS-Ocean. For each variation, performance with respect to various configurations of number of MPI tasks and OpenMP threads on a single Cori node is shown. Using explicit OpenMP directives to parallelize the element loops consistently outperforms both the approaches using pre-computed element distributions: naive static and depth-aware distributions.

We implemented these threading approaches within the routine `advection` in the ocean core in order to evaluate their performance. The generated results are shown in Fig. 4. In these experiments we observe that a straight-forward approach using explicit loop directives consistently performs equivalent to (or marginally better than) the latter approaches with pre-computed element distributions making performing any intelligent balanced distribution redundant. As we show in later sections of this paper, this is primarily due to the code being heavily memory bound. We will present detailed reasonings and validations on why being memory-bound makes any intelligent depth-awareness useless in order to improve performance of this code. Thus, for the remainder of this study we implement and consider MPAS-Ocean threading only via the explicit OpenMP loop directives.

B. Memory Initialization

In the baseline OpenMP threaded implementation, the complete ocean core within MPAS was threaded. This did not include the common MPAS framework which lies outside the ocean core, as mentioned previously in Section II. Since the memory allocation and initialization routines are also a part of the framework, and not ocean core, they were used as is. Therefore, all memory initialization through the use of these routines in the ocean core was single threaded. In our preliminary performance evaluations of the threaded ocean core, we observed that certain routines scaled poorly with increasing number of threads, making the overall performance worse in the hybrid mode, indicating the presence of Amdahl bottlenecks. This behavior is highlighted in Fig. 5 for a

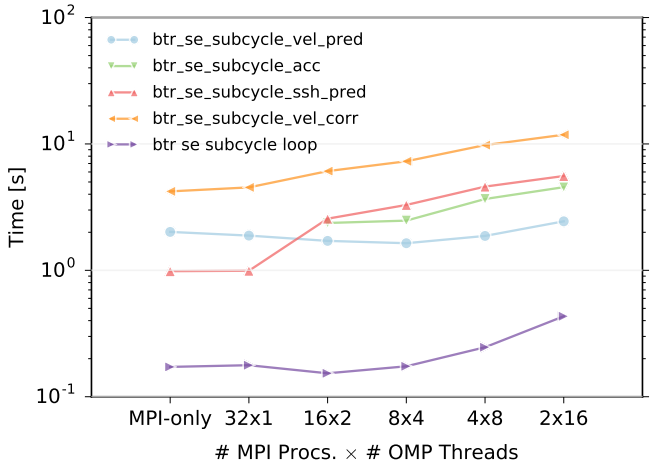

Fig. 5. Performance of a small set of compute functions. Several of the functions shown exhibit an exponentially degrading performance with increasing number of threads, indicating the presence of Amdahl bottleneck in the threaded code. Note that the y-axis is in log-scale.

small set of routines. Further investigation revealed that this behavior was primarily due to the use of these single threaded memory management routines, limiting the thread scaling. Furthermore, since the operating system uses a "first-touch" policy in allocating requested memory, initialization performed by a single thread put the entire buffer nearest to the one core executing this thread, resulting in NUMA effects creating additional performance impediment in certain configurations.

Since buffer initialization is necessary in this application, setting all buffer entries to default safe values, updating the memory management framework to address these performance issues was, hence, necessary to obtain any higher performance with the threaded ocean core. To remedy this impediment, we implemented threaded versions of these memory management routines. This significantly improved thread scaling, as shown in Fig. 6. Note, some routines, which make system calls to get pages from the OS, will always be slower. Moreover, as one cannot guarantee first touch in every call to the memory routines, since the actual buffer could have been allocated and initialized during application initialization, performance is sensitive to NUMA issues in the 1×32 configurations. Nevertheless, performance is now generally constant across process vs. thread configurations within a NUMA node, resulting in improved scaling. With the removal of this Amdahl bottleneck, the overall simulation performance improved by up to $2.5\times$ at high thread concurrencies.

C. OpenMP Thread Scheduling Policies

We next examine how various OpenMP thread scheduling policies affect the performance of MPAS-Ocean. The OpenMP standard [7] provides three basic scheduling policies, which may be selected by the programmer as a part of the loop directives, or at runtime when the `schedule(runtime)` clause is specified with the loop directives. These basic policies are: *static*, *dynamic* and *guided*. An additional parameter, `chunk_size` can be given with the static and dynamic scheduling, which specifies the granularity of the loop iteration chunks. The mapping from iteration count to a thread is predictive in the static scheduling, in which by default, the total iteration count is divided equally into n_t chunks, where n_t is the number of threads. A smaller chunk size results in a round-robin distribution among the threads until all chunks are assigned, and a size of one results in interleaving of threads with loop iterations. On the other hand, dynamic scheduling assigns iterations to threads in an on-demand basis, with a default chunk size of one. The exploration of chunk size is designed to explore whether the default data is amenable to constructive locality in a last level cache (whether adjacent threads can operate on spatially close or overlapping data) while the use of dynamic scheduling is designed to evaluate whether thread load balancing can be improved.

The performance results of these experiments with mesh 1, run with varying scheduling policies and chunk sizes, are shown in Fig. 7 for the Cori system. The different chunk sizes used are: default, 100, 10 and 1 for static, and 1 (default) and 10 for dynamic scheduling. It can be observed that the performance drops considerably for small chunk sizes, making the default dynamic and static with chunk size of one the worst performing policies in our case, while the default static and guided policies exhibit the best performance and scaling with increasing number of threads. A summary of the performance with respect to the scheduling policies and increasing number of threads per MPI process is shown in Fig. 8 as heat maps for Cori using 1, 2, 4, and 8 nodes. As a result of these experiments, we select static as the runtime for scaling experiments.

D. Application Scaling

In Fig. 9, we show strong scaling performance of the hybrid threaded implementation of MPAS-Ocean, for a various MPI process and OpenMP thread combination configurations, in terms of both simulation time and energy consumption. At low concurrencies, configurations with lower number of threads outperform the others significantly. However, at higher concurrencies, this trend slowly changes in favor of configurations with higher number of threads, resulting in the threaded versions offering significantly better scaling than the MPI-only version. This behavior is because performance scaling of configurations with higher number of MPI processes are bound by both, increased communication traffic and volume, and extra computations associated with the larger halo region volumes. Ultimately, the flexible hybrid solution allows users to tune the threads vs. process balance to identify optimal configurations – a process subtly different from the MPI-only or all OpenMP approaches.

E. Memory footprint analysis

A major advantage of incorporating threading is the reduction in the memory requirements of the code. In this section, we analyze the peak resident set size requirements for MPAS-Ocean in our experiments. These memory usage results are

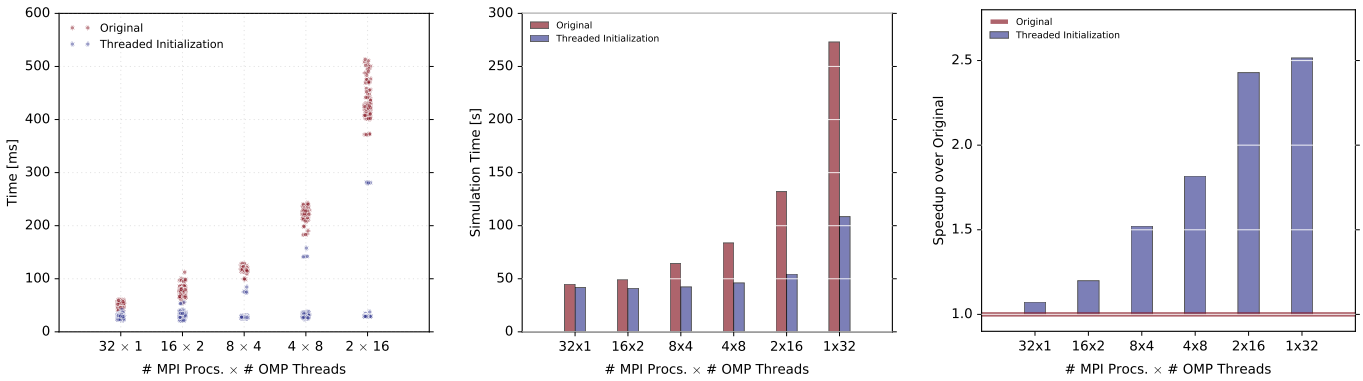

Fig. 6. Benefits of threading memory initialization routines: (Left) Detailed timings for the memory management routines called within the `equation_of_state` routine are shown. Data for all threads across all processes are shown where each data point represents a thread. An exponential decrease in performance is observed for the baseline single-threaded memory management routines when scaling with the number of threads. The performance of the optimized version of the code improves significantly with the use of threaded memory initialization routines. (Center) Total simulation time of MPAS-Ocean using the version with threaded initialization is compared with the baseline version, (Right) and the corresponding speedups are shown. Apart from improving the performance of the memory initialization routines, the overall application performance shows speedups of up to 2.5× at full threading. This performance boost is also an effect of improved data locality with respect to each thread. All these runs were performed on one node of the Cori system.

shown in Fig. 10. It can be observed that with increasing thread count (and decreasing MPI process count), the peak memory requirement is dramatically reduced at any given concurrency by up to an order of magnitude. This reaffirms our initial motivation to implement threading in this application.

Fig. 10 (right) shows the peak memory requirements per node with respect to the thread-process configurations on a node. In all cases a node operates on the same problem size with the same number of sockets and cores. As such, the figure quantifies the ability of hybrid MPI+OpenMP solution to dramatically reduce the memory requirements per node. For example, at 32 nodes, the memory requirements have been reduced by almost 5×. Beyond 32 nodes, memory requirements of MPI-only version have saturated while those for the hybrid implementations continue to decrease.

Intel Xeon Phi and GPU-accelerated systems rely on a two-tiered approach with a hierarchy of low-capacity, high-bandwidth memory backed by high-capacity, low-bandwidth memory. On a Knights Landing-based system like the Cori Phase II, there may be less than 16 GB of high-bandwidth memory per node. Simultaneously, the process concurrency in an MPI-only regime will increase to over 60 processes per node. As shown in Fig. 10 (left), the resultant 266MB per process limit will likely exceed the minimum memory requirements per process in MPAS-Ocean, and will thus not fit in the high-bandwidth level of memory. Conversely, the hybrid configurations can easily fit in high-bandwidth memory and will thus be able to effectively exploit its performance potential.

V. Vectorization

Vectorization provides the single-instruction multiple-data (SIMD) parallelism within each core on the compute processors. The available vector units on modern processors have increasingly wider widths, with common Intel Xeon CPUs, including Ivy Bidge and Haswell processors, containing 256-bit wide vector units, and Intel Xeon Phi series of processors containing vector units of 512-bit width. To gain a reasonable performance on such architectures, it is necessary to utilize these SIMD units effectively. Modern compilers are sophisticated enough to perform auto-vectorization and generate vectorized code without requiring a programmer to explicitly implement vectorization.

To facilitate efficient auto-vectorization, the code must satisfy certain conditions. Among them is that the loops to be auto-vectorized should not have backwards loop dependencies. For effectiveness, it is also desired that the data involved in these loops be stored contiguously in the memory to avoid excessive data gather/scatter operations. In our case with MPAS-Ocean code, since the vertical depths are represented by structured and contiguous data, the innermost loops over these vertical levels are the candidates for vectorization.

In Fig. 11, we show the observed performance of the various vectorized versions generated by the Intel compiler version 16 on the Cori system. The different vectorized versions shown are:

- Auto-vectorization completely disabled (with compiler flags $-$`no` $-$ `vec` and $-$`no` $-$ `simd`).
- Default compiler auto-vectorization with the base threaded code (with compiler flags $-$`vec` and $-$`simd`).
- Code containing the OpenMP SIMD directives added for the loops over the regular vertical levels (with flags $-$`simd` and $-$`no` $-$ `vec`).
- Compiler auto-vectorized code with the OpenMP SIMD directives included (compiled with flags $-$`vec` and $-$`simd`).
- Fully compiler auto-vectorized code with aligned memory (using flags $-$`align`, $-$`vec`, and $-$`simd`).

These experiments show that vectorization has almost no effect on the overall performance of MPAS-Ocean. This is a result of the primary performance limiting factor being

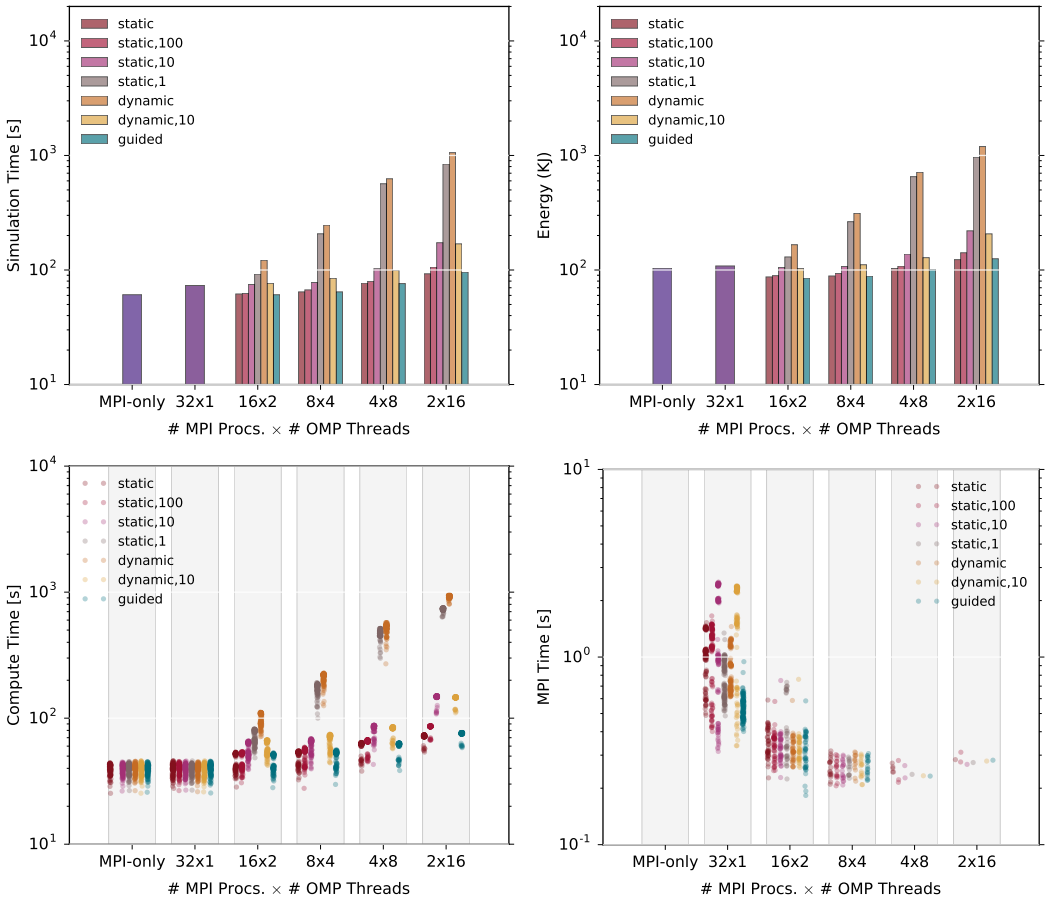

Fig. 7. Performance with different OpenMP thread scheduling policies on 4 nodes of the Cori system: (Top-left) Total simulation time of the hybrid MPAS-Ocean implementation is shown for seven scheduling policies: static, static with chunk size 100, static with chunk size 10, static with chunk size 1, dynamic, dynamic with chunk size 10, and guided. Data is shown for different configurations of MPI processes and OpenMP threads on each node, with number of threads increasing from left to right. (Top-right) Corresponding energy consumptions in KJoules is shown for each run. (Bottom-left and bottom-right) Detailed timings of each thread, representing a dot, are shown for sum of compute-only and MPI routines of the simulation are shown, respectively.

data movement, which is exacerbated by significant jumping in memory due to the unstructured nature of the data and buffer padding with maximum vertical depth. At low concurrencies, one would expect the time to be dominated by local computations. In order to understand why vectorization (even completely disabling it) has little effect on MPAS-Ocean performance, we first constructed a Roofline Model [8], [9], [10], [11] for MPAS running on Cori (see Fig. 12). We plot both the L1 and DRAM arithmetic intensities (flops/L1 bytes and flops/DRAM bytes, respectively) on this model. Although the L1 arithmetic intensity is very low at about 0.05, the L1 bandwidth is significantly high and, thus, doesn't constrain performance. Conversely, although DRAM arithmetic intensity is still low at about 0.2, the DRAM bandwidth on Cori is much lower and, thus, bounds the performance. On this system, with 256-bit vector units, the peak floating-point performance per node is roughly 1 TFlops/s. However, with vectorization disabled, the effective peak would drop to about 256 GFlop/s. Even though this is about 4× lower than peak, this flops ceiling (bound) on performance is still about 10× higher than the DRAM bound. As such, it should come as no surprise that for this heavily DRAM-limited code, vectorization has little impact.

VI. Conclusions and Future Work

Overall our study shows that the OpenMP threaded MPAS-Ocean code delivers better performance at high concurrencies when compared with the original flat MPI version. This indicates the importance of the use of shared memory parallelism as the core counts on modern processors increase. Our performance analysis indicates that memory accesses tends to dominate over the computations, making MPAS-Ocean a memory-bound code at low node concurrency. Future work will focus on improving the memory access patterns through data reorganization so as to allow better data re-use across the threads while improving cache utilization. In MPAS-Ocean, at high concurrencies, communication time dominates, and we plan to mitigate this overhead via communication hiding schemes through non-blocking data movement across the processes. This work is also paving the way to make the MPAS-Ocean code future-ready for many-core processors such as Intel's Knights Landing, which will drive the forthcoming

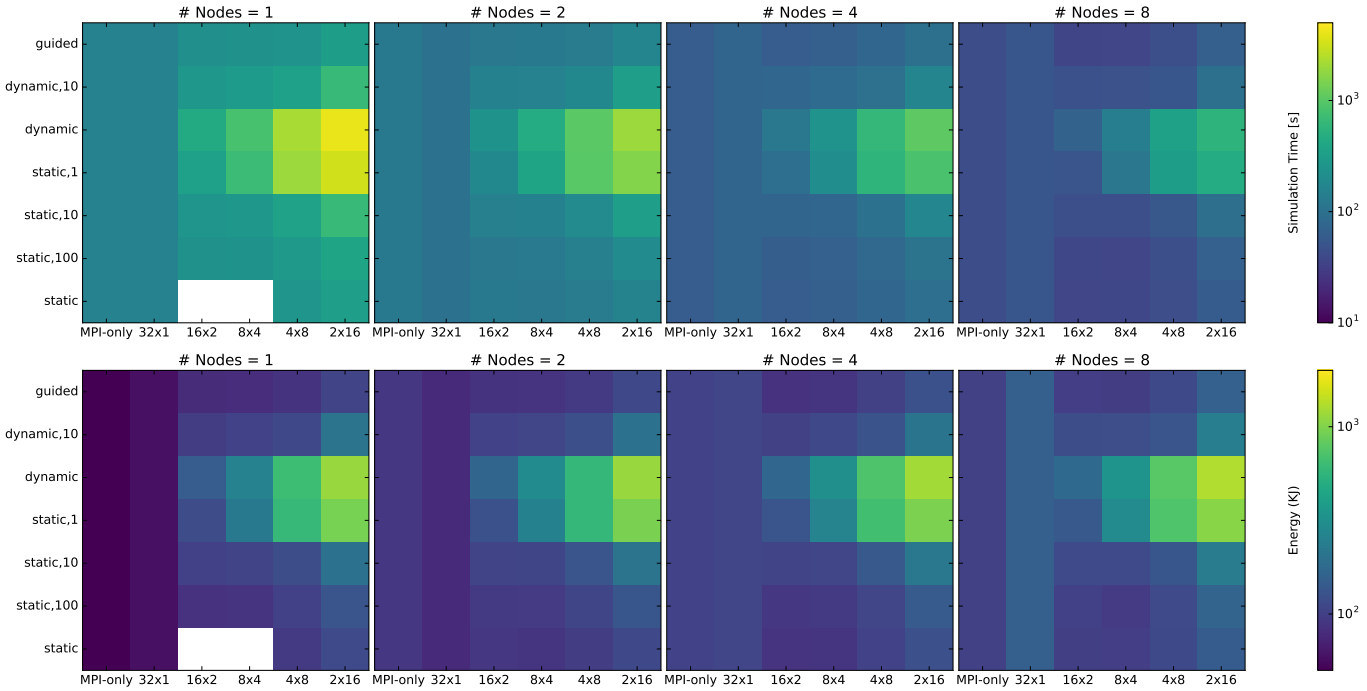

Fig. 8. A summary of the performance with different OpenMP thread scheduling policies and MPI×OMP configurations on the Cori system is shown as heat maps for 1, 2, 4, and 8 nodes. The top set represents the total simulation time in seconds, and the bottom set represents total energy consumption in KJoules.

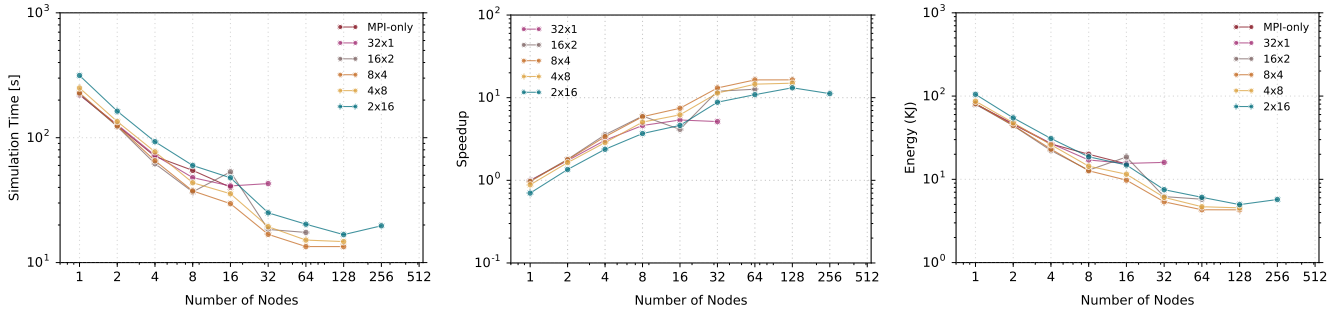

Fig. 9. Strong scaling with different MPI processes and OpenMP thread configurations on the Cori system with respect to increasing number of nodes. Each node provides a total of 32 cores. The total simulation times (left), corresponding speedups w.r.t. MPI-only on 1 node (center), and energy use (right) are shown for static scheduling.

Cray XC40 Cori phase 2 supercomputer at NERSC.

Acknowledgements

This research used resources in Lawrence Berkeley National Laboratory and the National Energy Research Scientific Computing Center, which are supported by the U.S. Department of Energy Office of Science's Advanced Scientific Computing Research program under contract number DE-AC02-05CH11231. Abhinav Sarje, Samuel Williams, and Leonid Oliker were supported by U.S. Department of Energy Office of Science's Advanced Scientific Computing Research program under contract number DE-AC02-05CH11231. Douglas Jacobsen and Todd Ringler were supported by U.S. Department of Energy Office of Science's Biological and Environmental Research program. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program.

References

[1] T. Ringler, M. Petersen, R. L. Higdon, D. Jacobsen, P. W. Jones, and M. Maltrud, "A multi-resolution approach to global ocean modeling," *Ocean Modeling*, vol. 69, no. C, pp. 211–232, Sep. 2013.

[2] H.-S. Na, C.-N. Lee, and O. Cheong, "Voronoi diagrams on the sphere," *Computational Geometry*, vol. 23, no. 2, pp. 183–194, 2002.

[3] J. Chen, X. Zhao, and Z. Li, "An Algorithm for the Generation of Voronoi Diagrams on the Sphere Based on QTM," *Photogrammetric Engineering & Remote Sensing*, pp. 79–89, Jan. 2003.

[4] A. Sarje, S. Song, D. Jacobsen, K. Huck, J. Hollingsworth, A. Malony, S. Williams, and L. Oliker, "Parallel performance optimizations on unstructured mesh-based simulations," *Procedia Computer Science*, vol. 51, pp. 2016–2025, 2015, international Conference On Computational Science, (ICCS 2015). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915012740

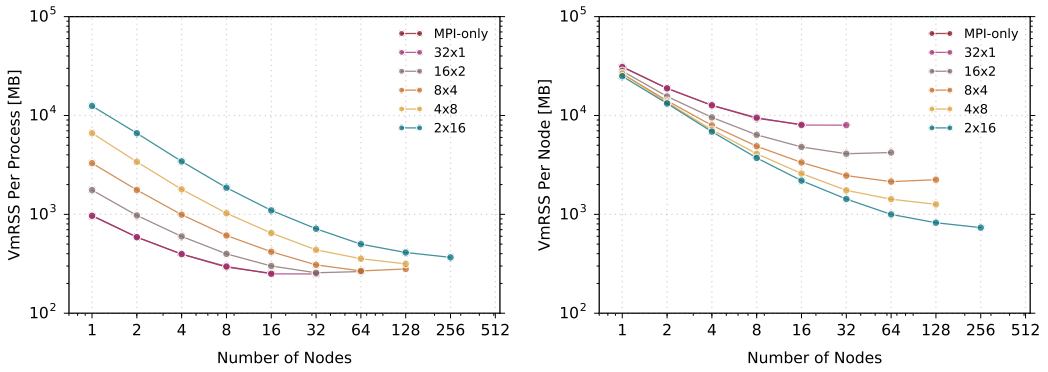

Fig. 10. Strong scaling memory usage per process (left) and per node (right) are shown.

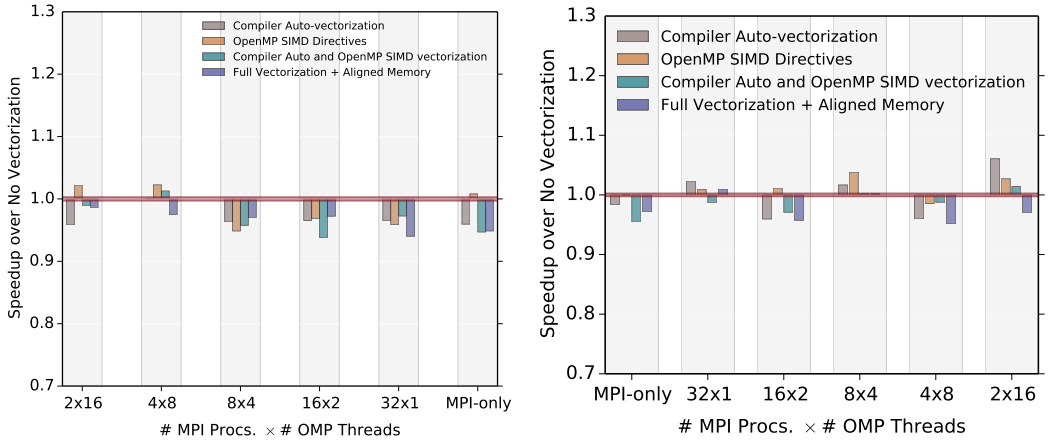

Fig. 11. A relative performance comparison of different vectorized versions of the `equation_of_state` routine (left) and `diagonal_solve` routine (right), with respect to no vectorization are shown. The different vectorized versions are: default compiler auto-vectorization (`-vec -simd`), code with OpenMP SIMD directives added to the loops (`-simd -no-vec`), auto-vectorized code containing the OpenMP SIMD directives (`-vec -simd`), and code with aligned memory (`-align -vec -simd`). It can be seen that vectorization almost no effect on the overall performance of this routine, indicating that data movement is the performance limiting factor.

[5] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System," *High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, May 2006.

[6] "Performance Application Programming Interface (PAPI)," 2016, http://icl.cs.utk.edu/papi.

[7] *OpenMP Application Program Interface, Version 4.0*, July 2013.

[8] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," *Communications of the ACM*, vol. 53, no. 4, pp. 65–76, Apr. 2009.

[9] S. Williams, "Auto-tuning performance on multicore computers," Ph.D. dissertation, EECS Department, University of California, Berkeley, December 2008.

[10] T. Ligocki, "Roofline toolkit." [Online]. Available: https://bitbucket.org/berkeleylab/cs-roofline-toolkit

[11] S. Williams, B. V. Stralen, T. Ligocki, L. Oliker, M. Cordery, and L. Lo, "Roofline performance model." [Online]. Available: http://crd.lbl.gov/departments/computer-science/PAR/research/roofline/

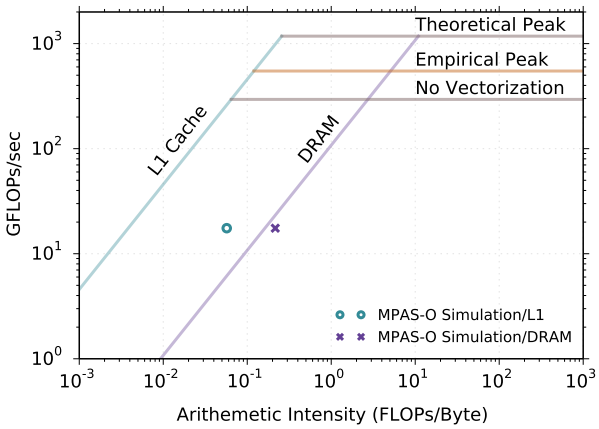

Fig. 12. A roofline analysis of the MPAS-Ocean code on the Cray XC40 Cori Phase 1. This plot shows that the MPAS-Ocean code is heavily memory bound since the achieved performance with respect to DRAM is almost at the peak DRAM bandwidth.