

# Evaluation of the MPAS Dynamical Core

Mark Govett, Jim Rosinski, and Tom Henderson

NOAA Earth System Research Laboratory

July 9, 2016

## Summary and Recommendations

MPAS-A is a modern global numerical weather forecast model, coded almost exclusively in Fortran. It lives on an icosahedral mesh, which mitigates the notorious “pole problem” found in earlier global models that utilized Gaussian grids or similar latitude/longitude grids. MPAS employs the C-grid with computations at the cell centers, edges and vertices. The code is well-written, well-documented, and the development team has made stable releases of the code and documentation publicly available at [github.com](https://github.com).

Coding strategy in MPAS is well-designed, with the computations of shared edge and vertex values limited to “owner-computes”. This strategy saves 2X the computations on edges and 3X the computations on vertices as compared to the situation where each cell sharing the edge or vertex redundantly computes the value. The recent implementation of OpenMP threading capability by the MPAS team gives slightly better performance than the pure MPI case. A good OpenMP implementation is critical to achieving good performance on the MIC and GPU fine-grain architectures, particularly when employing a high core count relative to horizontal resolution as might be necessary to achieve time to solution targets when run in forecast mode.

The code scales well in a strong-scaling sense of adding processors but keeping spatial resolution invariant, in fact showing super-scaling due to cache effects for some code regions at some core counts. An exception to this scaling behavior is the MPI communication, which is only slightly better than flat when moving to higher process counts at the same resolution. Adding vectorization and thread parallelism to the MPI message packing and unpacking routines, we were able to improve communications performance by up to 60%. However, communications scaling remained almost flat. Numerous possible user-code improvements to the communication are possible, and are described in detail in the section “Performance Profile and Scaling” below. The MPAS development team is currently exploring these solutions, one of which will potentially eliminate the need for a problematic I/O library (PIO) which sits on top of netCDF libraries `netcdf` and `netcdf4`.

Porting MPAS code to the GPU and MIC processors was straightforward. The entire model was ported and run on the latest generation MIC KNL processor. In addition, a standalone kernel routine, representative of the dynamics code in general, was extracted from the model to permit deeper examination and performance comparison. GPU performance results (2X slower than the CPU) were disappointing because they were significantly less than we observed with the NIM (1.7X faster than the CPU). This is likely due to a small number of vertical levels (55) in the kernel, and other factors suggested in the report. Intel’s Knight’s Landing processor (KNL) demonstrated significant speedup over the previous generation Knights Corner processor (KNC), largely due to fast memory (MCDRAM)

on the chip. NVIDIA's Pascal chip, expected in August 2016, is packaged with similar memory, that will likely give a 2-3X boost in application performance.

Based on our analysis, we offer the following recommendations:

- Improve portability of the PIO library or find an alternative
- Test on alternative compilers including Cray, IBM, PGI, etc.
- Improve scalability of inter-process communications
- Explore additional fine-grain optimizations including running a case with more vertical levels
- Explore using redundant calculations of edges and vertices to decrease inter-loop dependencies to improve fine-grain performance
- Further exploration of the potential benefits of dimensioning arrays via compile-time constants for all dimensions.

## Background

MPAS-A lives on an icosahedral mesh, with options to define refinement regions (higher resolutions) over portions of the globe of special interest. Our primary analysis focused only on the quasi-uniform resolutions. We initially used the publicly released MPAS-A (atmospheric) code (v4.0) for our analysis work. This version is a pure-MPI code.

The MPAS developers recently implemented OpenMP threading, and we were able to perform our runs with this hybrid OpenMP/MPI version of the code. OpenMP capability is critical in order to make optimal use of fine-grain architectures which enable it, such as Intel's MIC platform. We relied on a version of the code obtained from Michael Duda of NCAR. The packaged file which formed the basis of our analysis was `MPAS_20160325.tar.gz`.

Initial work on the MPAS dynamical core by NOAA/ESRL was done by Tom Henderson, who has since left NOAA. Much of his work, along with assistance by the MPAS development team at NCAR (primarily Michael Duda and Ryan Cabell), has been incorporated into the MPAS developer repository, and will eventually be migrated to the publicly released code. A summary of this optimization work on a single-threaded kernel is presented in Table 1 and Table 2 below. The meaning of each column is as follows: "IVDEP" is a compiler directive which was needed in some cases in order to get the compiler to vectorize the loop; "xAVX" is a compiler flag which indicates that the target architecture supports a certain level of vector instruction. "nopt" is a code optimization which hides the fact that Fortran pointers are being used in some cases, where the compiler may not be as aggressive as it otherwise could be; "const" is a code optimization whereby compile-time constants are used instead of run-time variables to define array sizes and loop ranges--compilers can sometimes generate more efficient code if these items are known at compile time; "nodiv" is a code optimization whereby floating point divisions were replaced by multiplications by reciprocal, since floating point divisions can be very expensive.

Further work was done on this "atm\_compute\_dyn\_tend" kernel. Initially, the routine only ran in single-threaded mode. In order to re-enable threaded capability (required for comparison between

CPU, GPU and MIC), we revised the embedded horizontal loops to iterate over the entire horizontal domain, with implied OpenMP synchronization at the end of each threaded loop replacing !\$OMP BARRIER statements in the original code. Motivation for this approach is described below, along with results of various further optimizations in the single-threaded code, and single-node parallel results.

MPIxOMP	Unmodified	IVDEP	xAVX	noptr	const	nodiv
<b>16x0</b>	0.63639	0.59718	0.54199	0.47944	0.45516	0.40898
<b>2x8</b>	0.81999	0.68193	0.61722	0.60083	0.44200	0.39884

Table 1: Raw data for various performance improvements to routine "atm\_compute\_dyn\_tend" as run on a single node of NCAR's "Yellowstone" machine, an Intel SandyBridge CPU machine with 16 cores per node. Reported times are for 10242 horizontal points, in wall clock seconds.

MPIxOMP	Unmodified	IVDEP	xAVX	noptr	const	nodiv
<b>16x0</b>	1.00	1.07	1.17	1.33	1.40	1.56
<b>2x8</b>	1.00	1.20	1.33	1.36	1.86	2.06

Table 2: Same data as above, but normalized by percentage speedup over the unmodified code. Speedup is cumulative reading left to right. In other words, speedup listed under "nodiv" includes all the other optimizations to the left of it.

In MPAS-A, some variables live on cell centers, some on cell edges, and still others on cell vertices. In this structure, individual cells share both edge and vertex values with other cells. There are distinct possible computational approaches to dealing with shared edges and vertices: They can either be redundantly computed by the cells that share them, or they can be computed just once, then shared by all cells that require the values. The former approach is somewhat simpler but wastes computation (computing the same thing twice or thrice), while the latter approach can be more complicated but saves redundant computations. MPAS-A takes the latter approach.

Since the extracted (single-threaded) kernel looped over the entire horizontal domain, we did not have the information at hand how to split up among participating threads the exact number of cells, edges, and vertices for each thread to loop over in order to still get correct answers. To solve this problem, we recoded the kernel in the following way. Code sections preceding !\$OMP BARRIER statements were made into individual subroutines. Then a !\$OMP PARALLEL DO statement was placed in front of each embedded loop over cells, edges, or vertices which had been designed to run in threaded mode. This approach has two distinct advantages: First, explicit barriers previously embedded within loops were replaced by the implicit barrier at the termination of loops adorned with !\$OMP PARALLEL DO. We find this approach more readable and understandable. The second advantage to replacing embedded barriers with implicit barriers at the end of loops is that the iteration count for individual threads need not be pre-computed; all calculations iterate over the full count of cells, edges, or vertices assigned to the MPI task. These code modifications demonstrated similar performance compared to the original code.

## Performance Profile and Scaling

Figure 1 below profiles the most expensive code regions in the MPAS dynamical core, as run on NOAA's Haswell-based supercomputer "theia" at a relatively coarse resolution of 40962 horizontal grid points

or roughly 111 km. Runtimes are presented for a 6-hour simulation. Profiling was accomplished by enabling the auto-profiling capability in the GPTL timing library, in which library start and stop calls are attached to compiler-generated “hooks” at function entry and exit points. Since it was not trivial to run with physics disabled, a dynamics-only profile was obtained by installing an MPI barrier after completion of the physics just prior to the dynamics loop. In addition, within the dynamics calculation physics tendencies are calculated in a routine “physics\_addtend”. Since this routine does not yet have OpenMP threading enabled, it is not included in the profile.

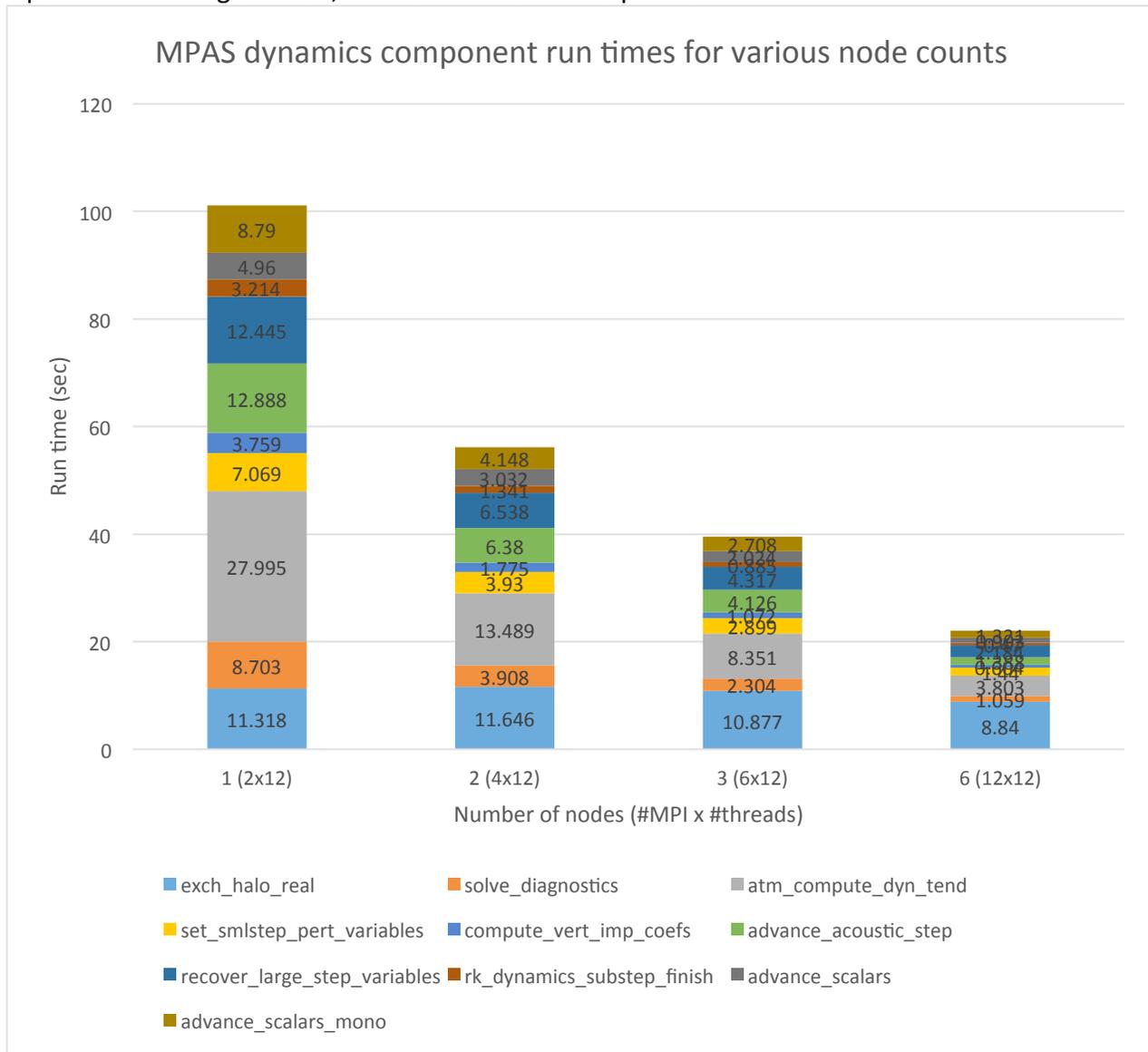


Figure 1: Timing profile for MPAS dynamics for various node counts on NOAA machine “theia” (Intel Haswell processors).

Computational scaling (all regions except “exch\_halo\_real” in the figure) is excellent from 1 through 6 nodes, with some regions achieving super-scaling for some node counts. Cache effects are the most likely cause of the super-scaling. If the scaling of all computational components were linear, a high-resolution run of 3.5 km would take the same amount of memory, and wall clock time per time step as the coarse-resolution test case presented here by employing 6144 nodes. Memory use at these

resolutions and node counts does not present a problem, as the high-water mark for the runs we did was only 13 GB when run on a single node.

Also in Figure 1, MPI communication of halo regions (“exch\_halo\_real”) shows some scaling (moving from 11.318 seconds on 1 node to 8.83 seconds on 6 nodes), but is nearly flat and therefore represents a potential area of concern regarding computational performance when moving to higher horizontal resolutions and higher node counts. We wondered if perhaps load imbalance in computational regions was being misrepresented as communication time. So we added artificial, timed, MPI barriers in key sections of code after computations and before communication. The result was that the 8.84 seconds taken by communication (rightmost entry in Figure 2) became about 8.3 seconds. This indicates some imbalance, but not nearly enough to change the overall conclusion.

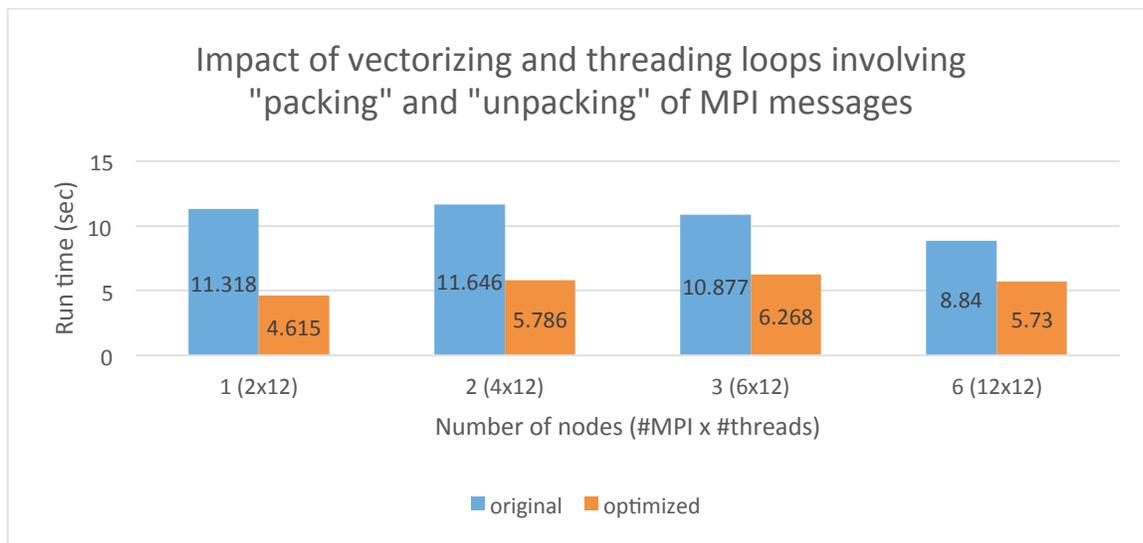


Figure 2: Impact of adding vectorization (`!$IVDEP` directive) and OpenMP threading to packing and unpacking of MPI messages on “theia”. Times labeled “original” are the same as in Figure 1 above. “Optimized” means threading was added to the outer loop, and the inner loop was vectorized.

We have spoken with the developers about improving MPI communication behavior, and they are aware of some modifications that can help. Specifically, one approach being worked on is similar to that taken with the NIM model developed at NOAA/ESRL. It involves strategic placement of points which will need to be communicated (halo points) in a way that better utilizes cache and may even eliminate the need to copy (pack) the data prior to communication.

Another advantage to improving the placement of halo points is to hopefully eliminate the need for an additional communication library (PIO) on top of already-required netCDF libraries pnetcdf and netcdf4. We have found the PIO library to be difficult to build, and some later versions of the library have both build problems and run-time problems not encountered in earlier versions. Also, we have found the situation of libraries on top of libraries, possibly on top of other libraries to be error-prone and difficult to maintain.

Also of note regarding MPI communication is that MPAS packs its messages into a contiguous address space prior to sending, then unpacks the messages after the data are received. In the original code, the

inner loop doesn't vectorize. Also, the outer loop involving packing wasn't threaded but can be with the addition of a `!$OMP PARALLEL DO` directive on the outer loop of the packing/unpacking code. The impact of adding vectorization and threading to these loops is shown in Figure 2 below. The total exchange time still doesn't scale very well, which will require further investigation. But by threading and vectorizing the packing and unpacking, the overall cost of halo exchanges drops dramatically. Depending on configuration, the speedup in the total cost of data exchanges is between 35% and 60%.

Other possible performance improvements to the MPI communication strategy include overlapping communication with computation where possible, and reusing exchange buffers rather than tearing down the linked list of exchange buffers and recreating every time step. We found a strange bug in a combination of the Intel Fortran 15.\* compilers and impi 5.\* MPI libraries where tearing down the linked list ran vastly slower (at least 3 orders of magnitude) than it should have. We understand from the developers that they are currently experimenting with overlapping communication with computation.

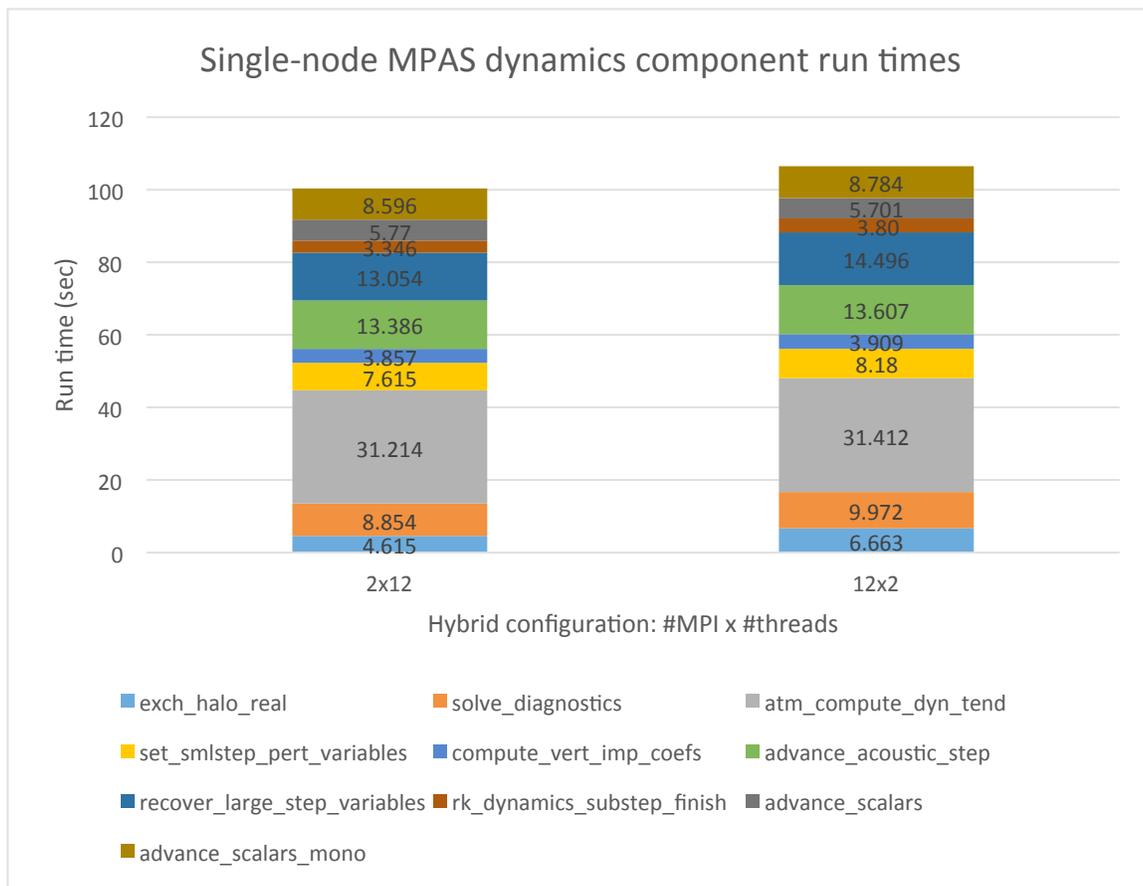


Figure 3: Single-node dynamics run times on Haswell processors (NOAA machine "theia") for mostly OpenMP (2 MPI x 12 threads) and Mostly MPI (12 MPI x 2 threads)

Figure 3 compares single-node MPAS dycore performance running in "mostly MPI" mode (12 MPI x 2 threads) vs. "mostly OpenMP" mode (2 MPI x 12 threads). Unlike Figure 1, the data for these runs was gathered with the MPI vectorization and threading optimizations discussed previously. Adding threading to MPAS did improve performance on theia, though the overall run times for both

configurations are close. The more highly threaded case (2x12) resulted in improved communication performance due to two primary reasons: The total number of halo points required to be exchanged between regions will be less because the size of each “owned” region will be greater. And since the size of each owned region is larger than the 12x2 case, the size of each MPI message sent/received will also be larger which minimizes aggregate latency due to fewer messages.

Though mostly-OpenMP provided only minimal computational advantage over mostly-MPI in MPAS, OpenMP capability is critical particularly for MIC architectures (KNC and KNL), where more numerous but slower processor cores put a larger strain on memory resources than traditional CPU architectures. A major component of the added overhead can be MPI library resources, so having fewer MPI tasks per node is an advantage on many-core systems such as MIC. A well-structured OpenMP implementation also makes porting to GPU a more straightforward process, since the OpenMP loops mark where fine-grain regions are implemented.

## Suitability for Fine-grain Architectures

### A. Full dynamical core

The MPAS development team has successfully ported the entire MPAS dynamical core and all supporting libraries to their pre-release KNL (MIC) system at NCAR. Initial results are promising. Shown below (data courtesy Michael Duda of the MPAS development team) in

*Figure 4* is a comparison of the best single-node performance results obtained on this KNL system as compared with the best CPU-based results we were able to obtain with the same code on NOAA’s “theia” machine (same data as in the left column in Figure 1 above). Theia has Intel Haswell processors, which are one revision older than the newest CPUs available from Intel (Broadwell), while KNL is brand new. But since both are modern processors, and the KNL is scheduled to move from 64 to 72 processor cores by its public release date, we consider this to be a fair comparison.

Notable in Figure 4 is that there is substantial performance gain on the KNL as compared with the Haswell generation CPU system. Though achieving this performance gain is predicated upon utilization of MCDRAM, a special type of extremely fast memory available from Intel only on their KNL systems. The NCAR KNL system has 16 GB of MCDRAM per node, which was more than sufficient to run an entire 111 km MPAS instantiation. Utilizing MCDRAM requires no code modifications if one wishes to place the entire executable in this fast memory. Code modifications in the form of compiler directives and special memory allocation calls are required if the user wishes to use only part of the fast memory for their run, while the rest can reside in standard, slower, memory.

The KNL results for MPAS are similar to what we have seen for other numerical forecast models. One very nice characteristic is that placing the entire executable in MCDRAM gives a dramatic speedup, on the order of 2-3X as compared with not using MCDRAM. Likewise, KNL generally gives a good speedup as compared with modern CPU-based architectures.

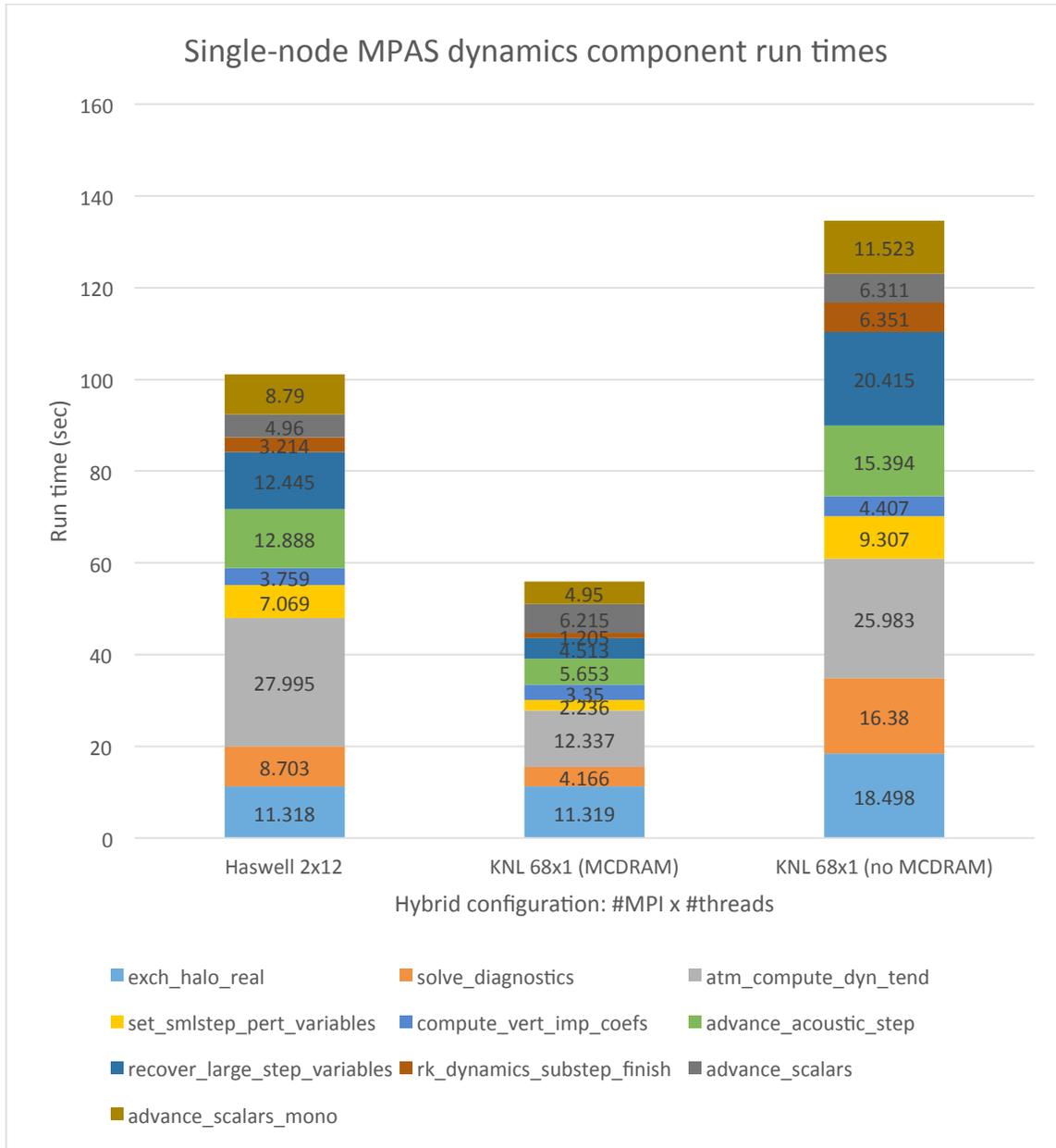


Figure 4: Single-node dynamics run times on Haswell (left), KNL + MCDRAM (middle), and KNL w/o MCDRAM (right). KNL data courtesy Michael Duda of the MPAS development team.

## B. Extracted Kernel

Early work by NOAA/ESRL investigated fine-grain suitability focused on generating a standalone test case for analysis. The subroutine chosen, “atm\_compute\_dyn\_tend” (same as described in Table 1 and Table 2 above), is representative of the code structure of the MPAS dynamics. As shown Figure 4 above, this routine consumes about 30 percent of the total runtime of MPAS dynamics. While the standalone test is useful for baseline comparisons with no code modifications, it also serves as a means to explore parallelization alternatives including code restructuring to improve fine-grain performance. The test

case had 10242 horizontal points with 55 vertical levels. The code was compiled and run using double precision floating point calculations.

## 1. GPU

OpenACC parallelization directives were added to the code to run on the GPU. The latest version of the PGI compiler (version 16.5) was used for the work. Use of the Cray compiler is planned. Both the **kernels** (*!\$acc kernels*) and **parallel** (*!\$acc parallel*) directive approaches were tried. These directives were wrapped in cpp pre-processing directives (PGI\_PARALLEL and PGI\_KERNELS) to easily switch between approaches. The **kernels** directive relies on compiler analysis to automatically determine loop level parallelism, data dependencies, and efficient use of memory. The **parallel** directive permits more user control over parallelization. Both approaches can be useful; **kernels** is simpler to get applications running quickly on the GPU, while **parallel** is often used to fine-tune performance.

Parallelization of the MPAS routine was trivial. Using the **kernels** approach, just two directives (*!\$acc kernels* and *!\$acc end kernels*) were needed for parallelization. Additionally, four *!\$acc loop* directives were used to identify variables private to GPU calculations. Depending on analysis by the compiler, private variables can use either the faster “shared” memory, or slower “local” memory. PGI diagnostics indicated the fast shared memory was used in all of the kernels where private variables were needed. Using the **parallel** approach, *!\$acc parallel* loop directives were placed immediately above each horizontal loop in the routine; 20 such directives were used. Additional *!\$acc loop* directives were used to identify *gang* and *vector* loops in the code help the compiler with analysis; these turned out to not be necessary as they did not improve performance.

Optimization of the routine included reducing the number of threads used (*vector\_length*) from the default 128 to 64 (GPUs want multiples of 32), to more closely match the 55 vertical levels used in the test case. This optimization gave almost a 2x improvement in performance. Adjustment of compiler flags gave further improvement including the use of “cc35” (compute capability v3.5) to target machine code specific to the Kepler hardware, and “maxregcount” to limit register use by the kernels. Other options were tried but did not improve performance.

Figure 5 compares the performance of the extracted kernel for a single, same generation dual-socket Intel IvyBridge versus a NVIDIA K40 GPU. CPU performance is shown using the PGI (16.5) and Intel (v15.0) compiler, where PGI is slightly faster. A newer version of the Intel compiler might show improvement. Runtimes using parallel and kernels directive were identical, indicating good automatic analysis by the PGI compiler. As indicated in Figure 5, GPU runtimes are 2X slower than the CPU times. Poor GPU performance (NIM used the same icosahedral grid, was 1.7X faster than the CPU on identical hardware) could be due to several factors. First, the number of vertical levels is only 55, which is not able to take full advantage of the GPU hardware (multiples of 32 is best), and is significantly less than needed to hide memory latency with other computations. A test case with 128 vertical levels is likely to significantly improve GPU performance versus the CPU.

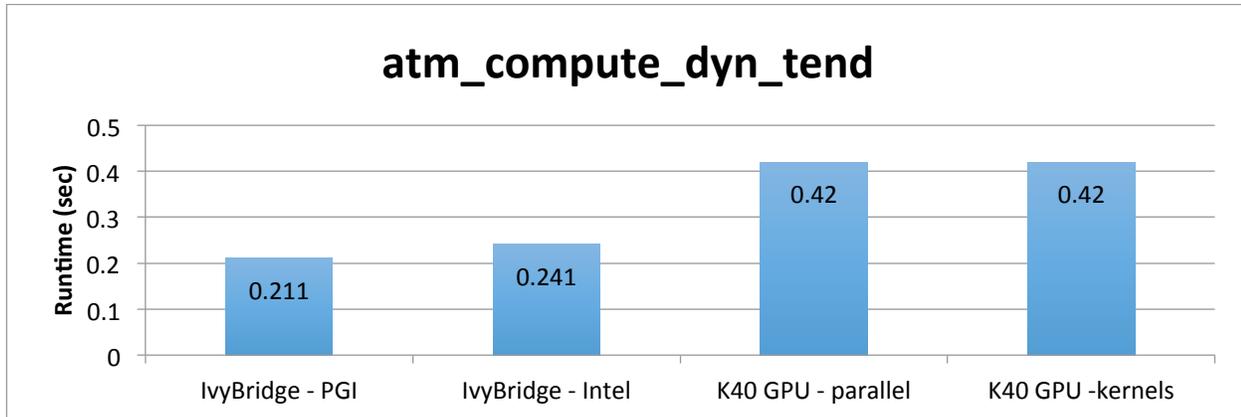


Figure 5: Performance of the standalone kernel on same-generation dual socket Intel IvyBridge and NVIDIA K40 GPU. Run times are shown for the CPU using the PGI and Intel compiler, and for the GPU using !\$acc parallel and !\$acc kernel directives for parallelization.

A second reason for poor performance may be due to the large number of GPU kernels generated. This is partially caused by interdependencies between 2D loops for edge, vertex and cell calculations that are inherent in the C-grid used by MPAS. Restructuring loop nests, fusion, and other techniques may result in larger kernels with more computations and fewer synchronization points. In the current regime, openACC parallelization generates 16 GPU kernels in the code, which required synchronization even when none was necessary. However, we were not able to measure significant overhead due GPU kernel startup and synchronization. Further investigation is warranted.

Finally, the GPU contains a relatively small amount of fast memory used for registers and shared memory that could significantly improve performance. GPU profiling showed that register pressure in some of the kernels limiting occupancy, a measure of concurrency on the GPU device. Further, GPU shared memory was lightly used in the MPAS kernels with opportunities to significantly increase performance if it can be exploited. Further investigation is needed to understand and overcome the performance issues noted here.

## 2. MIC

Figure 6 compares the performance of the same atm\_compute\_dyn\_tend kernel on similar-generation SandyBridge (CPU) and KNC (MIC) processors, as well as Haswell (CPU) and KNL (MIC) processors. The results demonstrate over a 3X improvement over the KNC. KNL's high-bandwidth memory does not give as much of a performance advantage as the full model shown in Figure 4.

We suspect that cache effects in the kernel are the main reason, noting that the horizontal grid in the extracted kernel was only 10242 horizontal points, while the full model runs were on a grid of 40962 points.

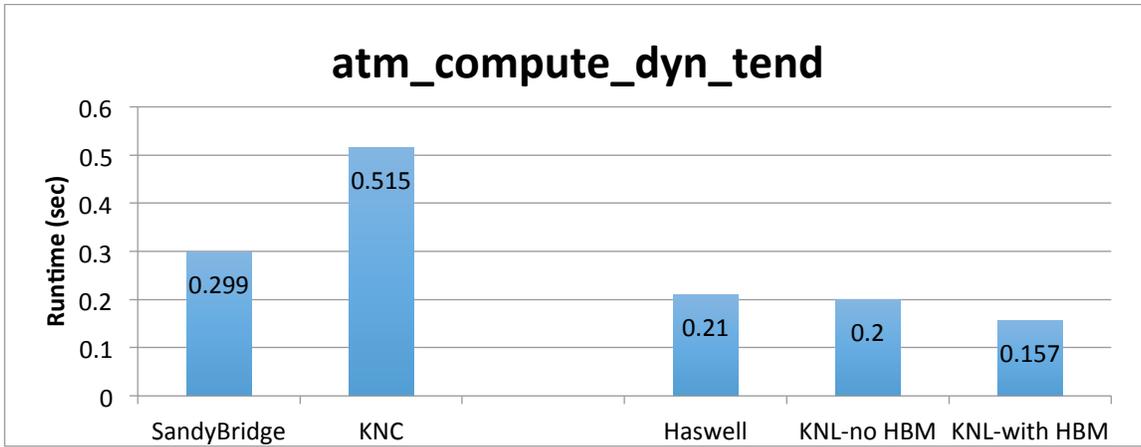


Figure 6: Performance of same standalone kernel as described in Figure 5 above, for same-generation CPU and MIC. LHS is SandyBridge and KNC. RHS is Haswell and KNL, where KNL time was measured with and without MCDRAM (HBM).