Preparing CESM for Exascale computing

John M. Dennis, Chris Kerr, Brian Dobbins, Richard Mills, Sheri Mickelson, Youngsung Kim, Raghu Kumar, Kevin Paul, and Allison Baker

Abstract Need an abstract here.

1 Scientific Methodology [2-3 pages, Dennis, Baker]

2 Algorithmic detail [2 pages, Dennis]

We focus our optimization efforts of two specific sections of the CESM code base. The Community Atmosphere Model (CAM) which typically consumes greater then 50% of the total cost of a typical CESM simulation. The CAM model is constructed of a large number of different code modules that are typically categorized into a dynamics and a physics modules. The dynamic core solves a set of partial differential equations which describe the fluid flow of the atmosphere, while the physics calculates all other properties. For example the physics modules would calculate the impact of solar radiation on aerosol particles and their chemical reactions with other atmospheric chemical species. In this section, we describe work performed to optimize the spectral element (SE) based dynamical core and the implict chemistry solver. The dynamical core, which can consume 30-88% of the total cost of the Community Atmosphere Model (CAM), has a key impact on the overall scalabilty of CAM, and CESM. For the dynamical core we focus on both increasing it's single threaded performance as well as increasing the amount of threads it can utilize. The implicit chemistry solver uses a ??? numerical method to solve a system of chemical

John M. Dennis
Computational & Information Systems Laboratory, National Center for Atmospheric Research, Boulder, CO 80307-3000 e-mail: dennis@ucar.edu

Richard D. Loft
Computational & Information Systems Laboratory, National Center for Atmospheric Research, Boulder, CO 80307-3000 e-mail: loft@ucar.edu

equations at each point in the atmosphere model. Depending on the exact scientific configuration of CESM, the cost of the implicit chemistry solver can vary from insignificant, to approximately 12% of the total time. While less important overall then the dynamical core, we include the implicit chemistry solver to illustrate the type of improvements that are possible for physic based modules.

2.1 Spectral element dynamical core

The National Center for Atmospheric Research's Community Climate System Model uses a High-Order Method Modeling Environment (HOMME) atmospheric dynamical. HOMME uses a spectral element method to discretize in the horizontal and a finite difference approximation [6] in the vertical. A continuous Galerkin finite-element method [7] is used for the spectral element method. The integrals used in the Galerkin formulation are computed from a Gauss-Lobatto quadrature rule within each element. HOMME decomposes each time-step into components and the equation, for a compressible fluid with hydrostatic and a shallow water approximations, can be written in terms of a vector U containing the prognostic state variables (velocity, temperature, and surface pressure) as: $dU/dt = F + D + A + T + R$ Where F represents the forcing from physics, D the dissipation, A the dynamics from the primitive equations, T the tracer advection, and R the vertical remapping of the mass and momentum variables. HOMME solves these equations in a time-split fully-explicit form. For time-steps involving the forcing (F) and dissipation (D) terms, a forward Euler time- scheme is used. The dynamics (A) and tracer advection (T) are computed using an N-stage Runge-Kutta time-scheme. The dynamics (A) computes the primitive equations prognostic variables. The tracer advection (T), is based on a finite-volume algorithm and advances the specific humidity, liquid water, ice variables, and additional tracer constituents. For advection, a vertical Lagrangian approach used [4] where the horizontal advection on Lagrangian vertical levels is followed by remapping (R) the mass and momentum variables back to the reference vertical levels at the end of the time-step.

2.1.1 General code Design

The HOMME dynamical core is written in FORTRAN 90 and utilizes a hybrid (MPI-OpenMP) programming model; all computations are performed with 64Byte reals and 32Byte integers. Since HOMME undergoes rapid evolution, implementation of optimization and parallelization changes needs to co-exist with other code modifications. Development for different architectures would require significant changes and produce multiple code versions which would be difficult to maintain. We have therefore chosen to develop a single unified version of HOMME for the Intel x86 architecture.

- Threading memory copy in boundary exchange

- Restructured data-structures for vectorization
- Rewrote message passing library/specialized communication operators
- Rearranged calculations in euler_step for cache reuse
- Reduced number of divides
- Restructured and aligned for better vectorization
- Rewrote and optimized limiter
- Redesigned the OpenMP threading

2.1.2 Single-core Optimizations

The single-core performance optimizations implemented included improving: vectorization, data locality, minimize computations, and compile-time specifications for loop and array bounds [3]. Each of these techniques has contributed to the overall single-core performance improvements in the code. The overall results of these optimizations are discussed in Section **??**.

The software design approach used in HOMME inhibited some optimizations techniques. The inner-loops, details of which are described below are written as (np,np) were collapsed and vectorized, but the loop index was not linearized as the compiler generates gather and scatters and not unit stride loads and stores. As the number of vector-registers on the AVX2 and AVX-512 has increased linearizion of the loop indices to generate longer vectors becomes of significant importance to performance.

The use of function statements throughout the code does impact performance. When inlining was forced with a compiler directive, the compiler did inline the function, however, the code generated was not as efficient as when the function was manually inlined. This was the result, not of the data copy across the function boundary, but from the optimizer performing more aggressive loop optimizations for the manually inlined code.

Finally, the compiler reports, provide information on details of alignment of data for each vectorized loop. Where the data was not aligned, the compiler flag (-align arraybyte64) and the OpenMP aligned directive does force data alignment. However, the directive is cumbersome to create for complex loop as all unaligned variables need to be defined in the aligned list. This creates a maintainability issue; there are also issues when alignment is forced as it can lead to incorrect answers. For these reasons, we have avoided forcing data alignment in the code.

2.1.3 Parallelization Strategy

The original scheme used parallelization over elements at a high-level and parallelization over loops at lower-levels. The revised scheme uses the same high-level parallelization over elements, however, the lower-loop level parallelization has been replaced with a high-level parallelization over the vertical level and tracers dimensions. The dimension chosen to parallelize is dependent on the region (dynamics, tracer advection, dissipation, or vertical remapping) of the code.

The revised scheme has several improvements over the original approach. These improvements include: a significant reduction in the number of threaded regions created; allows a greater number of threads to be assigned; allows replacement of MPI-Ranks with OpenMP threads which reduces the total MPI communication; provides greater flexibility in determining where the threads are assigned in the regions of the code; simplifies the scoping of variables in the parallel region as variables local to the routines, inside the parallel region, are private; minimizes wastage of thread resources as the parallelized over the vertical level and tracer blocks do not overlap so threads can be shared between these regions

The revised parallelization structure of HOMME is shown in Figure 1. Elements within the global domain are initially assigned with MPI-Ranks. Within the time loop, we maintain the original parallel structure over elements. The dynamics driver calls the dynamics, tracers, dissipation, and vertical remapping modules. These are parallelized over tracers or vertical levels; the blocking scheme chosen was dependent on the parallelism available in the module. In the dynamics, tracer and vertical remap modules, the code is blocked over tracers and in the dissipation module the code is blocked over the vertical levels. The only component module where we have maintained the original loop level-parallelism is in the 'compute_and_apply_rhs' module as dependencies in the vertical prevent blocking over levels. The code changes needed to implement the revised parallelization strategy were extensive. However, they could be made systematically which simplified the implementation process. Invocation of the blocked regions dynamics, tracers, dissipation, and vertical remap can be understood with a simple example:

```
call omp_nested(.true.)
!$OMP PARALLEL NUM_THREADS (num_threads_region) &
!$OMP& DEFAULT(SHARED), PRIVATE(hybrid_region)
  hybrid_region = config_thread_region(hybrid, region_type)
  call foo (hybrid_region, ...)
!$OMP END PARALLEL
call omp_nested(.false.)

subroutine foo (hybrid_region, ...)
    call get_loop_ranges (hybrid_region, loop_beg, loop_end)
end subroutine foo
```

Where the variable 'region_type' is name of the parallelism implemented in the module (element, level or tracer); 'hybrid_region' is a derived type that contains the mapping from the thread number to the starting and ending indices for each 'region_type'; '‘num_threads_region” is the number of threads assigned in the 'region_type'.

The declaration of the number of MPI-Ranks and OpenMP threads for the elements, tracer, and vertical is performed at run-time. The design of HOMME with

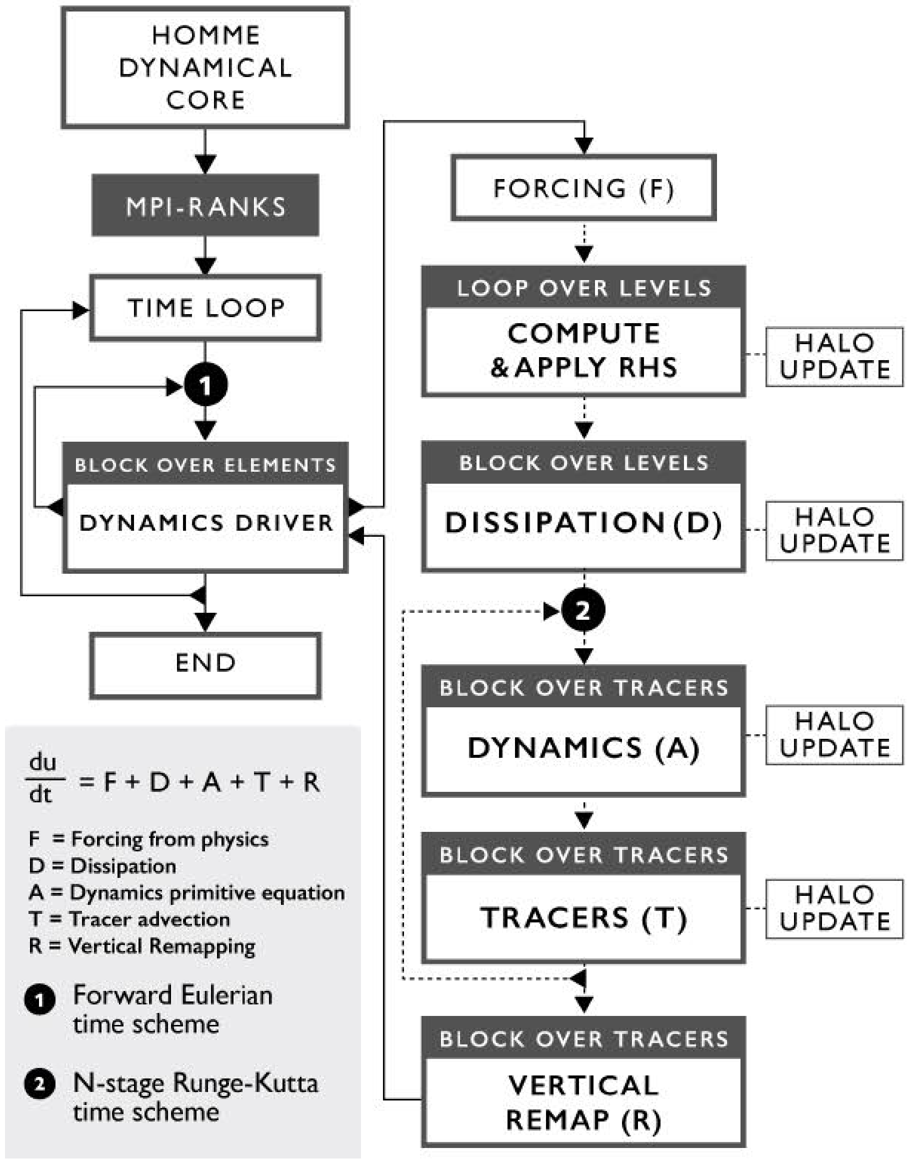

Fig. 1 The computational phases within HOMME.

inner indices (np,np) are known at compile-time which allows the compiler to generate efficient vectorized code with the limitations described in Section ??. The specification of the number of levels and tracers are made at run-time without impacting performance.

2.1.4 Data and Loop Structures

The data structures are dynamically allocated and maintain a global index within each MPI-Rank. Allocation of a generalized variable has the following ordering convention:

```
     variable_name[np,np,nlevel,ntracer,nelem]
```

Where np*np is the number of points in the quadrature grid, nlevel is the number oj vertical layers, ntracer the number of tracer variables, and nelem the number of elements per MPI-Rank.

At the highest level in the code, we have the loop structure:

```
do ie=nets,nete
    call dynamics_driver()
enddo
```

Where nets and nete are the starting and ending number of the elements in each MPI-Rank. The loop structure within the dissipation, dynamics, tracers, and vertical remapping modules are written as:

```
do q=qbeg,qend
 do k=kbeg,kend
    variable_name(1:np,1:np,k,q,ie) = ...
  enddo
 enddo
```

Where the element index 'ie' is defined above, 'kbeg, kend, qbeg, and qend' loop ranges are defined from 'get_loop_ranges' for the vertical levels and traces respectively and are evaluated for each thread number within the parallel region. The algorithm used to create these loop bounds is designed to minimize the load imbalance in the work that each thread performs. For the dimensions that are not blocked within the parallel region, they are simply defined as '1:nlevel and 1:ntracer'.

Prior to the blocked implementation, the loop bounds in the code, were written:

```
  do q=1,ntracer
     do k=1,nlevel
        variable_name(1:np,1:np,k,q,ie) =
     enddo
  enddo
```

A significant number of code changes were needed to create the blocked implementation. However, no changes were needed for the element region type as this was already supported in the original dynamical driver. The memory allocation for the variables remained unchanged. Changes were made to the halo-update communication routines so they were thread safe within the blocked levels, and tracer regions.

2.2 Implicit chemistry solver

The implicit chemistry solver within CAM uses a ??? method to solve an the system of chemical questions. It is based on the Mozart [**?**] chemistry pre-processor which takes as input a high-level chemical specification and generates Fortran source code. An independent set of equations is solved for each point in the atmosphere model. The chemistry pre-processor was initially designed in the 90's to generate vector code, and was later updated to generate scalar code. For the scalar code, the solution for a single grid point is calculated at a time while for the vector code, the solution for multiple grid points are calculated at once.

While the vector version of the implicit solver has the potential to accelerate the calculation though the use of instruction level parallelism there are several challenges that may impact the resulting speedup. For example, if there is a large difference in the number of iterations required to converge between different grid points, then the vector code will perform more calculations then the scalar version, because all points within the vector must converge. Additionally, there is the potential that the larger working set size for the calculation, approximately *veclen* where *veclen* is the length of the vector will have a negative impact on cache utilization.

The pseudo code for the implicit chemistry solver is illustrated in Figure **??**. Note the presence of an outer loop on line ?? over a block of one or more grid points, and an inner loop on line ??. While there are a total of four different routines is present in the iteration loop, each require a similar source code transformations. In section **??** we use the lu_slv subroutine as an example and describe each specific source code transformation.

```
ofl,ofu = set_block()
do while  ofu!=chnkpnts

  ! linear component of matrix
  lin_jac = linmat()

  do iter=1,itermax

      ! non-linear component
      sys_jac = nlnmat(lin_jac)

      ! factor the system matrix
      sys_jac = lu_fac(sys_jac)

      ! form f(y)
      forcing = imp_prod_loss()

      ! solve for the mixing ration at t(n+1)
      forcing = lu_slv(sys_jac,forcing)

      !check for convergence
      if(converged(forcing)) then
        break
      endif

   enddo
  ofl,ofu = set_block(ofl,ofu)

enddo
```

Fig. 2 Pseudo code for the implicit chemistry solver.

```
 do k=1,chnkpnts
   call lu_fac01(lu(k,:),b(k,:))
   ...
   call lu_slv01(lu(k,:),b(k,:))
   ...

subroutine lu_slv01( lu, b )

 real(r8), intent(in) :: lu(:)
 real(r8), intent(inout) :: b(:)

 b(17) = b(17) - lu(17) * b(16)
 b(19) = b(19) - lu(20) * b(18)
 ....
```

Fig. 3 LU solve for original scalar version of the code (s00)

```
    real(r8) :: lu(chnkpnts,nzcnt)
    real(r8) :: b(chnkpnts,clscnt4)

    call lu_fac01(ofl,ofu,lu,b,chnkpnts)
    ...
    call lu_slv01(ofl,ofu,lu,b,chnkpnts)
    ...

  subroutine lu_slv01( ofl, ofu, lu, b )

    integer, intent(in) :: ofl,ofu
    real(r8), intent(in) :: lu(:,:)
    real(r8), intent(inout) :: b(:,:)

    do k=ofl,ofu
      b(k,17) = b(k,17) - lu(k,17) * b(k,16)
      b(k,19) = b(k,19) - lu(k,20) * b(k,18)
      ....
    enddo
```

Fig. 4 LU solve for original vector version of the code (v00)

```
    real(r8) :: lu(cknkpnts,nzcnt)
    real(r8) :: b(chnkpnts,clscnt4)

    call lu_fac01(ofl,ofu,lu,b,chnkpnts)
    call lu_slv01(ofl,ofu,lu,b,chnkpnts)

  subroutine lu_slv01( ofl, ofu, lu, b, chnkpnts )

    integer, intent(in) :: ofl,ofu
    real(r8), intent(in) :: lu(chnkpnts,nzcnt)
    real(r8), intent(inout) :: b(chnkpnts,clscnt4)

    do k=ofl,ofu
      b(k,17) = b(k,17) - lu(k,17) * b(k,16)
      b(k,19) = b(k,19) - lu(k,20) * b(k,18)
      ....
    enddo
```

Fig. 5 LU solve for modified vector version of the code (v01,v02).

```
    use dim_mod, ONLY: veclen
    real(r8) :: lu(cknkpnts,nzcnt)
    real(r8) :: b(chnkpnts,clscnt4)
    real(r8) :: lu_blk(veclen,nzcnt)
    real(r8) :: b_blk(veclen,clscnt4)

    lu_blk  = lu(ofl:ofl+veclen,1:nzcnt)
    b_blk   = lu(ofl:ofl+veclen,1:clscnt4)

    call lu_fac01(aveclen,lu_blk,b_blk)
    call lu_slv01(aveclen,lu_blk,b_blk)
    ...

  subroutine lu_slv01( veclen, aveclen, lu, b)

    integer, intent(in) :: ofl,ofu
    real(r8), intent(in) :: lu(veclen,nzcnt)
    real(r8), intent(inout) :: b(veclen,clscnt4)

    do k=1,aveclen
      b(k,17) = b(k,17) - lu(k,17) * b(k,16)
      b(k,19) = b(k,19) - lu(k,20) * b(k,18)
      ....
    enddo
```

Fig. 6 LU solve for final vector version of the code (v03)

3 Programming Approach [1 page, Dennis]

4 Software Practices

4.1 General practices [0.5 pages, Dennis]

4.2 CESM Ensemble Verification [1.5 pages, Baker]

The Community Earth System Model (CESM), like most climate simulation codes, is large and complex. Further, the CESM code is continually evolving to accommodate recent science developments, to port to new HPC platforms requested by the climate community, and to prepare for future (e.g., exascale) machine architectures, for example. Given its ongoing state of development, quality assurance via software verification is particularly critical for CESM to both detect and reduce errors, thereby ensuring user confidence in the simulation code and its output. Verifying the correctness of a change or update to the CESM hardware/software stack (e.g., a new compiler flag, a difference machine, a code modification) is trivial when the simulation results after the update are bit-for-bit (BFB) identical with the original results. At issue is when the new results are no longer BFB with the original results. The chaotic nature of climate model simulations ensures that this occurrence is commonplace in the CESM development cycle, as tiny code modifications or a compiler change can easily result in non-BFB solutions (e.g., see [5]).

When the CESM simulation results have changed due to modification(s) to the hardware or software stack, determining whether the difference is significant (i.e., "climate-changing") is key. In particular, we consider a modification to be admissible if it is statistically indistinguishable from the original results. Until recently, making such a determination was a computationally expensive and subjective task that required climate expertise (e.g., running and comparing multi-century simulations). However, the CESM Ensemble Consistency Test (ECT) tool introduced [1] had formalized and simplified an important aspect of CESM quality assurance and is now regularly used by the CESM software engineers when porting to new CESM-supported machines and releasing code updates, for example. Further, because CESM-ECT is objective in nature and accessible to model developers and users, the tool has proven its utility in detecting errors and providing rapid feedback to software developers, thus boosting model confidence. In particular, the capability to determine consistency between simulation results that are not BFB provides the much-needed flexibility, for example, to pursue more aggressive code optimizations and utilize heterogeneous execution environments, both of which are critical in the path to exascale.

As its name implies, the CESM-ECT tool uses an ensemble-based approach to determine climate consistency (i.e., statistical indistinguishability). In particular, an ensemble of "accepted" simulations that represent the same earth system model allows us to gauge the natural variability of the models climate by providing a qualitative measurement of variability with which to compare future simulations. The CESM-ECT issues a pass or fail for new non-BFB CESM output (e.g., from a new machine) by comparing it to the variability represented by the accepted ensemble and determining whether it is statistically distinguishable. The CESM-ECT is a suite of tools tailored for individual CESM component, and, at present, ECT modules have been developed and released for the Community Atmosphere Model (CAM) component (CAM-ECT) [1] and ocean component (CESM-POP) (POP-ECT) [2]. Note that ECT modules can in some situations detect errors from other components (e.g., the example in [1] where CAM-ECT detects an error in the sea ice component).

While all CESM-ECT modules rely on ensembles, the underlying testing algorithms may be surprisingly different due characteristics of the respective models. For example,CAM-ECT characterizes the ensemble's distribution by performing principal component (PC) analysis on the global area-weighted means of 120 CAM variables from an ensemble of 151 one-year climate simulations (that differ only in an initial round-off level perturbation to the atmosphere temperature), and, from that, creates a distribution of accepted PC scores. CAM-ECT then compares three one-year runs with a modification (e.g., run on a new platform) with the accepted ensemble distribution. If more than a threshold number of PCs are outside of the distribution, then the new run is deemed to be inconsistant, or statistically distinguishable from the original (see [1] for further details). Because CAM and POP have differing characteristics in term of dynamics, spatial variability, and timescales, the CAM-ECT approach is not suitable for POP-ECT. In contrast, POP-ECT evaluates ensemble means and deviations for five independent diagnostic variables in a spatial manner, resulting in a characterization of ensemble distribution at each grid point (see [2] for further details). Developing modules for the other components is work in progress.

While CESM-ECT has greatly contributed to quality assurance for CESM, at present its scope is limited to coarse-grain verification. In other words, while CESM-ECT can readily indicate that a problem exists by issuing a "fail", an easy means of identifying the root cause of the failure is not provided. For example, for a failing test, CAM-ECT reports the PCs that differ from the ensemble distribution by more than a specified threshold amount. Because each PC is a linear combination of all 120 CAM variables, tracing the failure to a specific variable is non-trivial (e.g. see Section 6 in [5]) and identifying the error in a complex code with more than one million lines can be daunting. At least two additional capabilities are needed to more easily trace a CESM-ECT failure to its root cause. First, a utility is needed to identify potential problematic CESM kernels based on information from the CESM-ECT failure. Second, once the kernel(s) have been identified, a utility is needed to extract kernels from CESM (along with its relevant data) to render the debugging task reasonable. While we have not begun work on the first tool, a tool that fulfills the second capability is described in the following section. Finally, we note that our latest efforts have been focused on the development of an ultrafast version of CESM-ECT, which only requires nine model steps (equivalent to 4.5 hours in model time) and which has shown promising preliminary results.

4.3 KGen: Fortran Kernel Generator [2 pages, Kim]

KGen is a Python tool that automatically extracts a partial code out of a large Fortran application and converts it into a standalone/verifiable/executable kernel. Working with a kernel, instead of original large application, has several benefits in various software engineering tasks. For example, cycle time for debugging could be reduced dramatically as kernel generally takes much shorter time in compilation and execution. Particularly, kernel is an efficient vehicle for communicating between collaborators from various disciplines including application developer, compiler engineer and hardware developer. NCAR has generated kernels from several weather/climate models using KGen. The kernels can be downloaded from https://github.com/NCAR/kernelOptimization

Automatic kernel generation

Using KGen is a simple one-step process which reduces the amount of time and resource that might be needed if kernel generation is done manually. Figure 1 shows a workflow of using the tool.

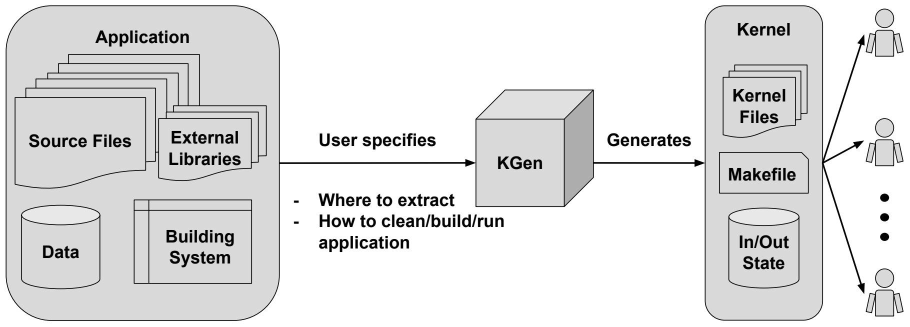

Fig. 7 KGEN workflow. Color not allowed in book, excessive space below image.

Many scientific applications are large and complex. Depending on their configuration, it may takes more than an hour to complete compilation/execution. Instead, the size of generated kernel is generally between 1K to 100K source lines and it may take less than a minute to compile and run. Furthermore, a kernel does not have any dependencies to external library. Therefore, it could be easily built/run on multiple different machines. Once generated, kernel can be easily usable by collaborators.

Since a kernel is a just another software that can be compiled and run independently, its usage is not limited to a particular case. For example, There are several cases that KGen-generated kernel has been used successfully including:

Performance optimization: Several performance-critical parts of CESM were extracted using KGen and distributed to multiple performance engineers who independently optimized the generated kernels. Optimized kernels are collected and applied to CESM. Validating Float-point calculation among different platforms: The same CESM simulation on two different system showed different simulation result. A suspicious part of code was extracted using KGen. After large number of trial-and-errors on the kernel, it is found that the root cause was a FMA(Fused-Multiply-Add) instruction. Compiler bug report: A part of code having a compiler bug is extracted. A compiler bug is reported with the kernel so that the bug could be reproduced easily on compiler vendor site. Porting to GPU: A part of radiation physics code is extracted and shared with an external research team for porting it on GPU. Private benchmark for procurement: One of time consuming part of CESM is extracted and inserted in a benchmark suite for one of recent NCAR procurement. A kernel generation example KGen distribution contains several examples. This section briefly steps through one of the examples to show how to generate a kernel using KGen. First, please make sure that your system meets following prerequisites: Linux OS, Python (higher or equal to 2.7 but less than 3.0), Make build utility, Cpp C preprocessor and Strace system call tracer.

Downloading KGen: run following Git command on your local directory. ¿¿¿ git clone https://github.com/NCAR/KGen.git

Moving to an example directory: This example shows how to extract a region of Fortran code. Please read README to check if you need any modification for your system setting. ¿¿¿ cd ./KGen/example/simple-region

Extracting a kernel: run make in the example directory. The make command actually invokes KGen command shown below.

5 Benchmarking Results

5.1 Description of computational platforms [1 page, Kumar]

Intel: Haswell Configuration:

Hardware: Dual Socket Intel Xeon processor E5 Two 2.3 GHz, 16 Cores/Socket, 72 Threads/Node. Each core has its own L1 and L2 caches, with 64 KB (32 KB instruction cache, 32 KB data) and 256 KB, respectively; there is also a 40-MB shared L3 cache per socket.

Software Stack: impi/5.1.3.210, cray-mpich/7.4.0, intel/17.0.0.042, cray-netcdf/4.3.3.1

Compiler Flags: -O3 -AVX2 -fp-model fast

Intel: Xeon Phi Configuration:

Hardware: Intel Xeon Phi processor 7250 68 Cores/Node, 1.4GHz, 272 Threads/Node, MCDRAM flat memory mode. Each core has its own L1 and shared per tile L2 cache, with 64 KB (32 KB instruction cache, 32 KB data) and 512 KB respectively.

Software Stack: impi/5.1.3.210, numactl/2.0.11, intel/17.0.0.042, cray-netcdf/4.3.2

Compiler Flags: -O3 -xMIC - AVX512 -fp-model fast

5.1.1 HOMME Performance Results

We evaluate the impact of the threading changes described in Section 2.1 using two different configuration of the HOMME dynamical core. The *perfTest* configuration uses 26 vertical levels (plev=26), and 25 tracers (qsize=25) and is similar to how the default configuration of CAM-SE. The *perfTestWACCM* uses 70 vertical levels (plev=70) and 135 tracers (qsize=135) and is comparable to the default configuration used by the Whole Atmosphere Community Climate Model (WACCM) [**?**]. HOMME uses a cubed-sphere computational grid, and horizontal resolution is indicated by the number of spectral elements on a side of a cube (ne). CAM-SE is typically run in production at ne=30 which corresponds to grid points approximately every 100 km. High resolution simulations with grid points approximately every 25 km (ne=120) typically require very large core counts for an extended periods of time. For example ASD simulation [**?**] executed CAM-SE at 25 km resolution on 21,600 cores of Yellowstone in 20?? for ?4? months.

The performance of HOMME is strongly influenced by the number of spectral elements allocated to each hardware core. The total number of spectral elements is

Property	Yellowstone	Laramie	Cori Phase I	Cori Phase 2	HPCFL
# of nodes	4,356	72	1,630	9,304	1
processor type	Intel Xeon E5-2670, Sandybridge (SNB)	Intel Xeon E5-2695-V4, Broadwell (BDW)	Intel Xeon E5-2698-V3, Haswell (HSW)	Intel Xeon Phi 7250 Knights Landing (KNL)	Intel Xeon Phi 7250 Knights Landing (KNL)
sockets per node	2	2	2	1	1
cores per socket	8	18	16	68	68
cores per node	16	36	32	68	68
total # of cores	72,576	1,296	52,160	632,672	68
base frequency (GHz)	2.6	2.1	2.3	1.4	1.4
L1 D+I cache	32 + 32 KB per core	32 + 32 KB per core	32 + 32 KB per core	32 + 32 KB per core	32 + 32 KB per core
L2 cache	256 KB per core	256 KB per core	256 KB per core	1024 KB shared by 2 cores	1024 KB shared by 2 cores
L3 cache	20 MB per socket	45 MB per socket	40 MB per socket	none	none
DRAM per node	32 GB	unknown	128 GB	96 GB	96 GB
MCDRAM	none	none	none	16 GB flat	16 GB flat
RAM type	DDR3-1600	unknown	DDR-2133	DDR4-2133	DDR4-2133
Vector Length (bits)	256	256	256	512	512
fp vector units	?1?	?1?	?1?	?2?	?2?

Table 1 Computational platforms.

calculated by the formula total_elm = 6 * ne * ne. For the ne=120 ASD simulation a total of 86,400 spectral elements were used resulting in the assignment of 4 elements per core. For the purposes of our evaluations, several different horizontal resolution configurations are used, (ne=4, 8, and 30). The ne=4 resolution has a total of 96 total spectral elements, while the ne=8 and ne=30 have 384, and 5400. We use the ne=4 and ne=8 configurations to measure the impact of our modifications on either a single node or small number of nodes. The ne=30 configuration is to measure the impact of our modifications at much larger node counts.

The execution time seconds/model day and the computational cost for both the original and optimized code is provided for the ne=4 configuration on Yellowstone is provided in Figure 9. Note that both the number of nodes used and the number of elements per hardware core is provided on the bottom and top x-axis on Figure 9. The left y-axis corresponds to the execution time in seconds per model day, while the right y-axis corresponds to the computational cost. The lines in Figure 9, which are solid and dashed with diamond data points that correspond to the execution time clearly illustrate the reduction in execution time. Figure 9 clearly illustrates that non only does the optimized code reduce the execution time, but it also allowed HOMME to scale to larger core counts. In particular, while HOMME was previously limited to a single element per hardware core, It is now possible to obtain speedup for this configuration out to $1/4$ element per hardware core. The plain dotted line and dotted line with diamond data points indicate the computational cost relative to the original code on a single node. These cost lines allow for an easy comparison of the impact that these optimizations may have on how HOMME is used. While the optimized version of HOMME will certainly reduce the cost of simulations on a fixed number of cores, it also introduces the possibility to use more hardware for a fixed computational cost. For example if we draw a horizontal line from the host to run HOMME using 6 elements per hardware core from the original code (plain dotted line) to the optimized code (dotted line with diamonds) it is apparent that is is now possible to increase the amount of hardware used from 1 node to 3 nodes without any increase in the computational cost. The ability to efficiently utilize more hardware without increasing computational cost is a important advance for the optimized code.

configuration has a total of 96 spectral elements, while our -degree grid has a total of 86,400 spectral elements. For our 2048-node production runs, we use a total of 64 execution threads per node which results in 2 or 3 spectral elements allocated to each 32K hardware core. The left panel in Figure 5, which has an x-axis that varies from 6 to spectral element per hardware core, is a configuration of the SE dynamical core that has 26 vertical levels and 25 tracers and is similar to a standard CAM configuration. We refer to this configuration of the SE dynamical core as CAM-like. The right panel in Figure 5, in which the x-axis varies from 6 to 1/8 spectral elements per hardware core, is a configuration of the SE dynamical core that has 70 vertical levels and 135 tracers and is similar to a standard configuration of the Whole Atmosphere Community Climate Model (WACCM). This particular configuration of the SE dynamical core we refer to as WACCM-like.

The solid blue line (labeled orig) corresponds to the version of the SE dynamical core used in CESM1, while the solid red line (labeled opt) corresponds to the version of the SE dynamical core used in CESM2. A significant reduction in execution time for the dynamical core is apparent for both the CAM-like and WACCM-like configuration. It is also important to note that good speedup is even achieved on problem sizes with less than a single spectral element per core. The ability to efficiently run the SE dynamical core on problem sizes that are smaller than a single spectral element per core is absolutely critical to being able to successfully utilize the increasing on-node parallelism that Xeon Phi offers. We have also included dotted blue and red lines which indicate the cost of the dynamical core relative to the CESM1 version using 6 spectral elements per hardware core. These cost lines allow for an easy comparison of the impact that these optimizations may have on science objectives. Recall that our current Mira jobs have a maximum of 3 elements per hardware core. If we draw a horizontal line from the cost to perform the CESM1 version of the SE dynamical core in the CAM-like configuration (left figure) using 3 elements per hardware core, it is apparent that we can utilize 3 times the amount of hardware cores with only a very modest increase in computational cost using the new SE dynamical core. Specifically, instead of using 3 elements per hardware core we can scale to 1 element per hardware core. A similar analysis for WACCM-like configuration reveals that instead of being limited to 3 elements per hardware core, we will be able to scale to element per hardware core, enabling the use of 12 times the number of hardware cores.

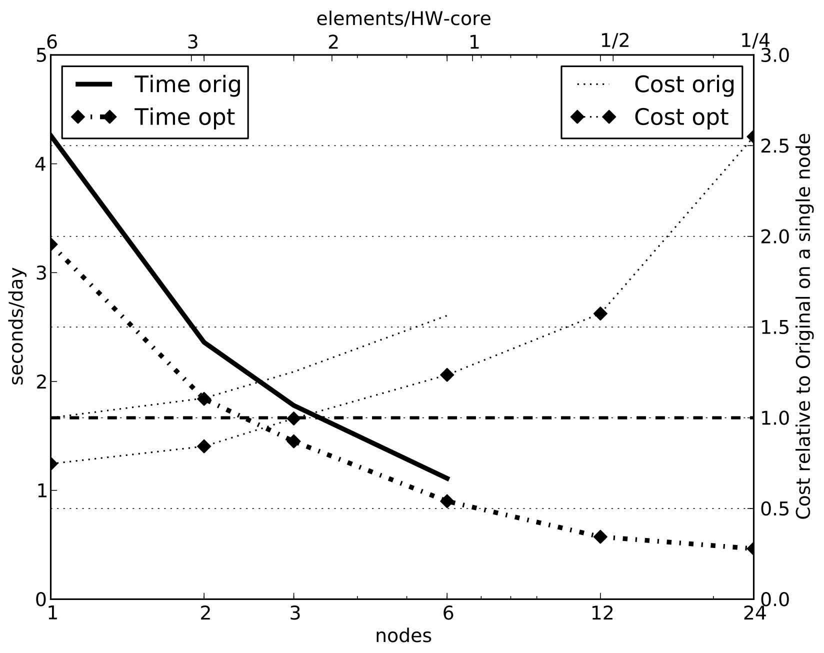

Fig. 8 Strong scaling for HOMME in perfTest configuration (ne=4, plev=26, and qsize=25) on 1-24 nodes of Yellowstone.

q Because of our IPCC-WACS project we have gained early access to single nodes of Knights Landing (KNL) hardware we are also able to provide timing results of the original and optimized SE dynamical core. Note that when run outside of CAM the dynamical core is also known as High Order Methods Modeling Environment or HOMME. Using this 68-core single socket system, we have been able to compare the performance of HOME on both the CAM-like and WACCM-like configurations to other single node systems. Note that for this comparison we configure HOMME to use 384 spectral elements. We choose a larger HOMME configuration for this comparison to avoid the load-balance issues that the small configuration would create on the KNL node. Figure 6 contains the timing for the CAM-like and WACCM-like configurations on the left and right side respectively. Results are provided for the following nodes: a 16-core Sandybridge (SNB2), a 32-core Haswell, (HSW2), and a 68-core KNL node. Note that in the case of the KNL node only 64 of the 68 cores are used. It is interesting to note that while the HSW2 node is still faster than the KNL node for the CAM-like configuration, the KNL node is significantly faster for the WACCM-like configuration. We suspect that this is the result of the on-package high-bandwidth memory that KNL provides.

We believe that the combined impact of both the scalability improvements within the SE dynamical core and the improved architectural features that a Knights Land-

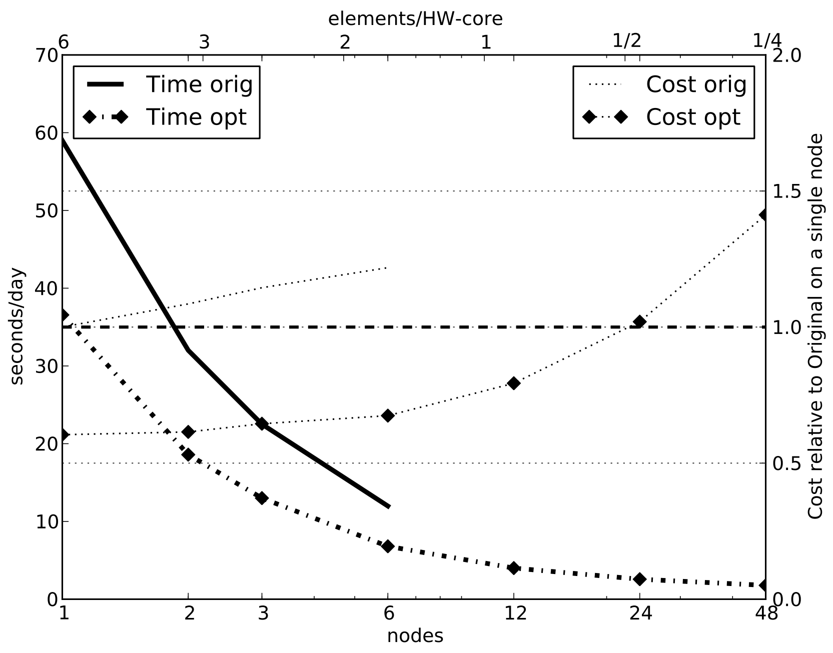

Fig. 9 Strong scaling for HOMME in perfTestWACCM configuration (ne=4, plev=70, qsize=135) on 1-48 nodes of yellowstone.

ing or Knights Hill based system offers will enable CESM2 to efficiently utilize 172,800 hardware cores for a single CESM ensemble member.

5.2 *Chemical pre-processor [3 pages, Dennis and Mickelson]*

The first task was to generate the proper code to optimize. This involved finding a version of CESM, along with a model configuration and resolution that was able to run both the scalar and vectorized versions of the code generated by the chemistry preprocessor. CESM model version cesm1_4_alpha07c was chosen with model configuration FWTC4L40CCMIR1 at 2 degree resolution (FV 1.9x2.5). This model configuration uses WACCM with Troposphere, Stratosphere, Mesosphere, and Lower Thermosphere (TSMLT) chemistry. The CESM/CMIP6 atmospheric chemistry runs will also include the Modal Aerosol Model 4 (MAM). This configuration was not used for this project because the code was still in flux and was not ready in time for this project. Because our goal was to optimize the chemistry preprocessor, our hope was that the optimizations we did at the tested configuration will be able to optimize all chemistry code generated by the tool.

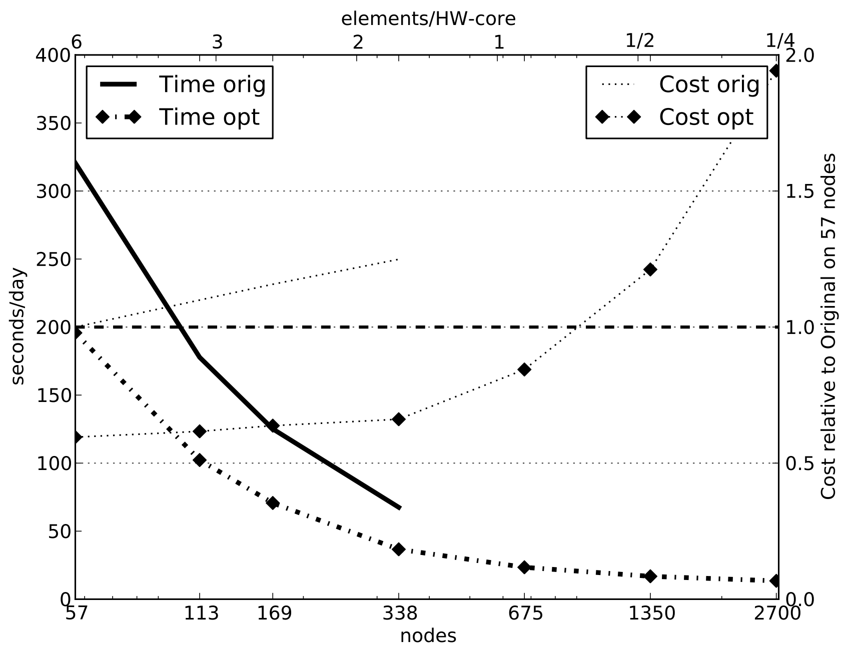

Fig. 10 Strong scaling for HOMME in perfTestWACCM configuration (NE=30, PLEV=70, qsize=135) on 57-2700 nodes of yellowstone.Need extra data points for larger core counts.

After a stable chemistry configuration was found, we generated a KGEN kernel [6]. KGEN allowed us to create a standalone-testing kernel. Once configured, it captured the Fortran code and the input and output arguments needed to run this section of the code. It also incorporated a testing environment that allowed us to test different optimizations and test to see if the answers were changed. This allowed us to experiment with different optimization techniques without the need of running CESM and provided a quicker turn around.

Two KGEN kernels were generated, one for the original scalar version and one for the vectorized version. The versions of the code were checked into a GitHub repository and code optimization work was checked into the repository as branches.

It is also important to get better results on Intel?s Haswell and KNC processors. The Haswell processor is similar to the Broadwell processor that will be installed into NCAR?s next production machine, Cheyenne [3]. CESM will also be expected to run efficiently on the Knights Landing (KNL) processor. This will be important as Cori Phase II goes into production at NERSC with KNL processors.

The first step was to get base timings for both the scalar and vector versions of the kernel. During this process it was discovered that the answers changed in both the scalar and vector version when the code was ran on the Haswell and KNC. It was found that the Fused Multiply-Add (FMA) instruction set was causing the answers

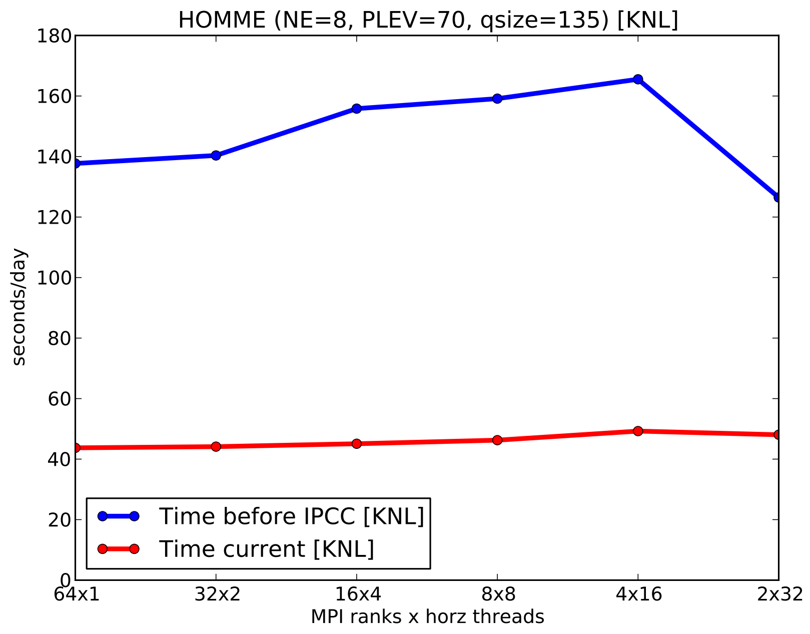

Fig. 11 Execution time for HOMME in perfTestWACCm configuration on a single node of Knights Landing.Redo with updated version of HOMME. Color is not allowed in book. Replace line graphs with bar graphs.

to differ. The answers matched when the code was built with the ?-no-fma? compiler flag. Since it was important to track if any of the optimization work changed answers, this flag was used in all builds. The base timings without optimizations are shown rows 1 and 2 in Table 2. The out of the box vector code out performed the scalar version on Sandy Bridge, but took longer to run on the other two architectures.

Our next step was to understand the code better and to determine how the code was or was not being optimized. This was done by building with the compiler flag ?-qopt-report=5? and by running with Intel Advisor [7]. Intel Advisor is a profiling tool that is able to evaluate a code?s performance. It is able to inform the user why performance is poor and provides suggestions on how to improve the performance. Both Intel Advisor and the optimization report stated that the compiler was performing no vectorization. This was because several of the array sizes were declared dynamically. Since the compiler did not know the sizes of the arrays, it did not know how to vectorize them.

In order to force the code to vectorize, we needed to declare the array sizes correctly. This was done by removing the dynamic declarations and by replacing them with the variable sizes. Code vectorization by the compiler was then confirmed within the optimization reports. The timing results are shown in row 3 in Table 2

(?Vector w/ Better Declarations?). This improved the speedup on Sandy Bridge by 1.38 and on Haswell by 1.35 over the original scalar kernel. This improved the KNC performance by reducing the time to ? of the original vector time, though better performance was still seen in the KNC scalar code.

Once the code was vectorized, our attention turned to improving the memory usage. As a simple test, we decided to modify the vector length. The original vector length had been set to 13 to match the number of columns in the model. The code then looped in groups of 13 until the total number of points, 858, was solved for. 858 is the total number of columns (13) times the number of levels (66). As a test, we decided to set the vector length to 858, the total for this loop. The timing results are in row 4 in Table 2 (?Vector w/ Vector Length = total # of points?). While this provided us with performance improvement across all three architectures, it also increased the cache misses because of it?s larger memory footprint.

	SNB	HSW	KNL w/DRAM	KNL w/HBM
s00	88.5	59.0	276.4	231.3
v01(vlen=858)	105.4	135.9	197.7	60.2
v02(vlen=64)	46.0	42.7	151.5	56.1
v03(vlen=16)	35.9	23.7		
v04(vlen=16)				

Table 2 Time in miliseconds for chemistry implicit solver kernel. Need v03 and v04 results.

In order to try to reduce the memory footprint, we tried adding ?Distribute_Point? directives to the file mo_lu_factor.F90, the most time consuming portion of the code. The DISTRIBUTE_POINT directive is used to divide up a large ?DO? loop into several smaller loops. When it is placed above the loop, the compiler tries to determine how to break up the loop. Placing the directive inside the loop forces the compiler to break the loop at that point [8]. The file, mo_lu_factor.F90, consists of twenty subroutines, each having one ?DO? loop that ranges in size from 100-600 lines. As a starting point, the directives were added every five lines within the loops. The results are shown in row 5 in Table 2 (?Vector w/ Distribute_Point to break up loops?). This increased the performance by about 0.1 on Sandy Bridge and Haswell and by about 0.01 on KNC.

We next wanted to try reducing the memory footprint through cache blocking. The easiest way to do this was to adjust the vector length variable. As discussed earlier, the original value was 13 (the number of model columns) and it had been changed to 858 (the number of columns times the number of levels) during optimization testing. We believed this larger memory footprint would be an issue as we scaled from 1 rank per node to 16 on Sandy Bridge and 72 on Haswell. The red bars in Figures 1 and 2 confirmed this. As the number of tasks was increased on a rank, it took longer to run because we were running out of space in cache. In both cases, the vector time took longer than the scalar time (the blue bars in Figures 1 and 2) when it was fully scaled, causing us to loose all of our previous optimization work.

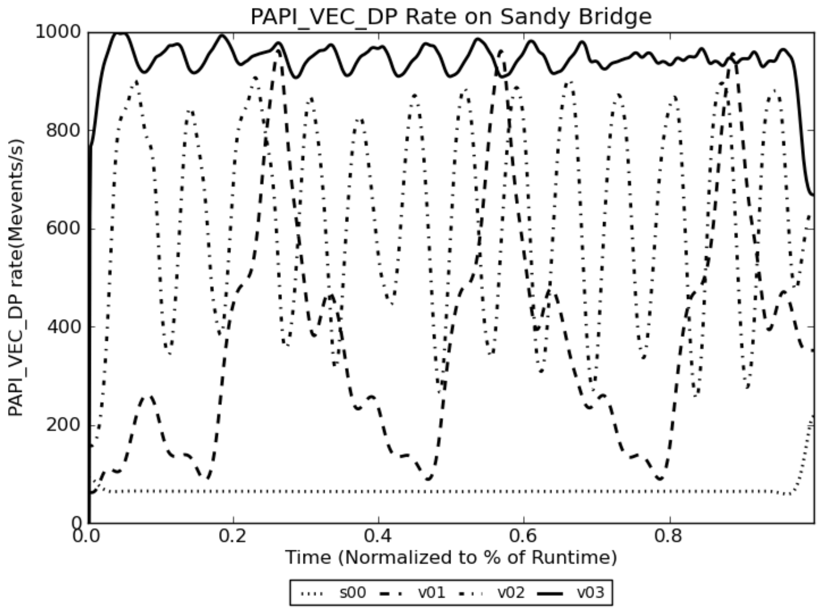

Fig. 12 A plot of the fraction vector instructions on Haswell for several different versions of the implicit chemistry solver.

In order to find the optimal value, we devised an experiment where the large arrays were systematically halved until our run times were not increasing with MPI task counts. This allowed us to find the largest size that would fit into memory and scale across the all cores. Through testing, we found that the optimal value was a vector length of 64 on Sandy Bridge. The improvements for both Sandy Bridge and Haswell are shown in the green bars in Figures 1 and 2. It was noted that the scaling on Sandy Bridge was slightly better than what was seen on Haswell. At full scale, the speedup over the scalar version was 1.92 on Sandy Bridge and 1.36 on Haswell. A separate test of a vector length of 32 was performed on Haswell, which yielded a speedup of 1.58 over the scalar version. This will have to be studied further to determine if the different architectures require different vector lengths or if there is another cause for this.

Setting the vector length to 64 also increased the performance of the code. This increased the performance on Sandy Bridge by about 0.5, 0.4 on Haswell, and it had little affect on the KNC. This brought the total increase in performance to 2.35 on Sandy Bridge and 2.01 on Hawell, our target optimization amounts. This is shown in row 6 in Table 2 (?Vector w/ Vector Length = 64?).

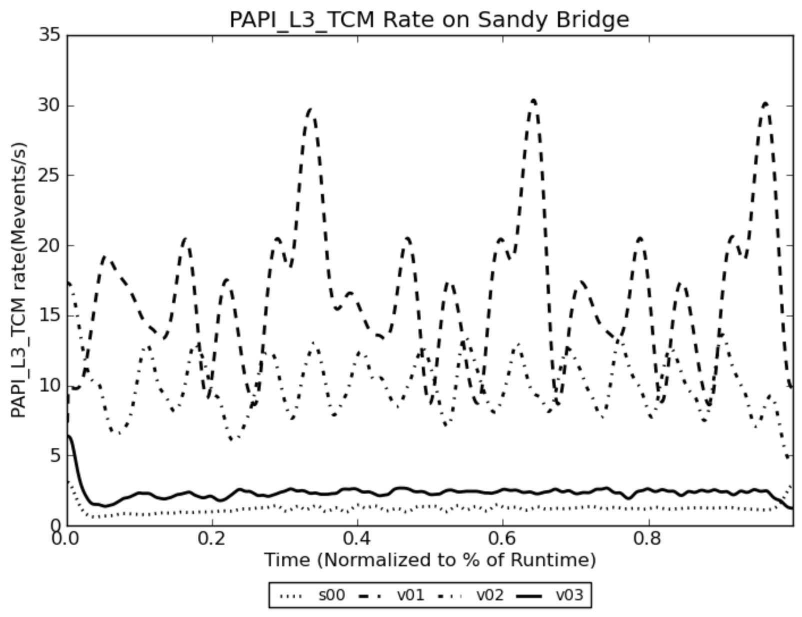

Fig. 13 A plot of the total L3 cache miss rates on Haswell for several different versions of the implicit chemistry solver.

5.3 Other CAM modules [1 page, Kumar]

binterp is an interpolation routine used in CESM under the CAM module. binterp is a subroutine accepting a little more than ten arguments and has roughly a hundred lines of code including comments. The lines of code included three doubly nested loop and several branching statements both within and outside the nested loops. Thus, the default execution on Sandybridge processor took about a few hundred nano seconds. Since the routine was called thousands of time, optimizing the kernel was essential. On profiling the code, it was observed that none of the loops vectorized due to branching statements within the loop. Moreover, more than 70The optimized binterp, using intel 15.0.3 compiler, was found to provide 3.17x and 1.31x speedup on Sandybridge and Haswell architectures respectively.

Radiation includes a number of routines, which compute cloud optical properties using CAM method within CESM. The radiation code is divided into two parts based on the solar spectrum as the longwave (LW) and shortwave (SW) radiation code. Each division of the radiation code has three algorithmic parts a) Stochastic array generator, b) Gas optics and c) Solver. The radiation code collectively has more than ten thousand lines of code including. Static analysis of the code showed that a)

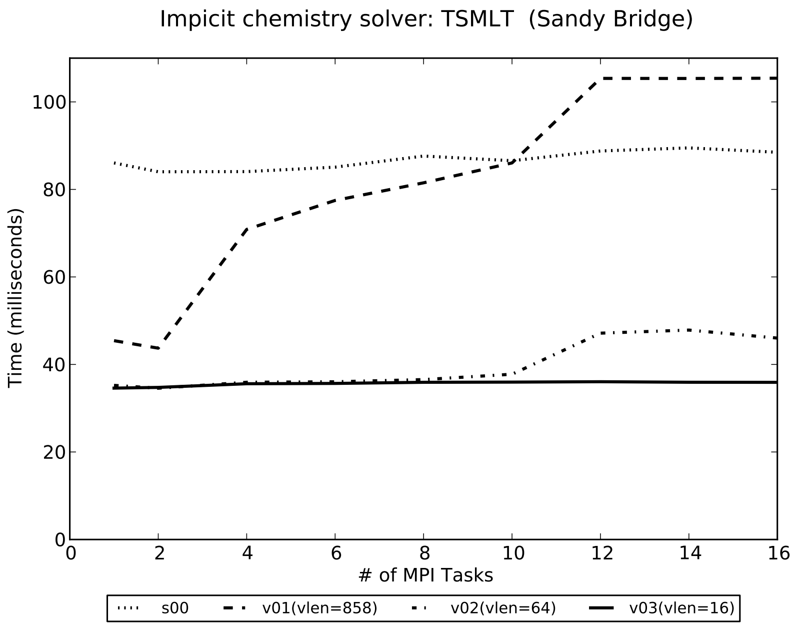

Fig. 14 Weak scaling of several different versions of the implicit chemistry solver on Sandybridge FIXME: v03 data is missing. Color not allowed in book.

large number of loop independent computations were repeated within loops, b) several unused copies of arrays were created, and c) no cache blocking strategies were used. Profiling of the code revealed that none of the main loops vectorized in the radiation code due to branching statements within the loops. Steps adopted for optimization: a) Loop independent operations were pushed outside the loop b) Unused variables were removed c) To facilitate cache blocking, the loops had to be pushed one to two levels lower, which required promoting of variables to higher dimensions and re-coding of routines to accommodate loops d) Loops were split to separate out branches, facilitating vectorization e) Redundant division operations were replaced with reciprocal multiplications For the compilation, additional flags/directives were added to align arrays, force vectorization on the loops and modify precision models. The optimized radiation code, using intel 15.0.3 compiler and MKL 11.2.3, was found to provide 1.62x speedup on Sandybridge architecture.

The optimizations performed on these modules collectively reduced the cost of CAM by approximately 4.7

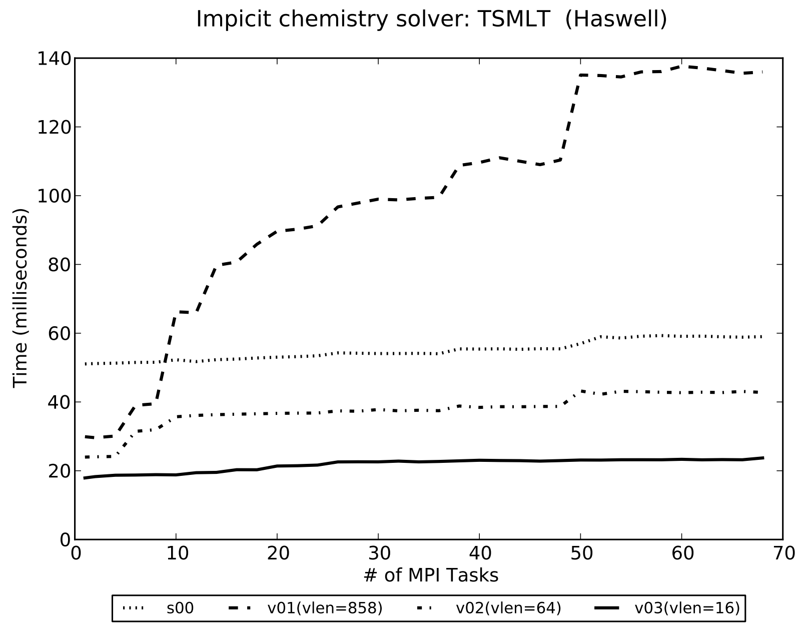

Fig. 15 Weak scaling of several different versions of the implicit chemistry solver on Haswell. FIXME: v03 data is missing. Color not allowed in book.

6 Accelerating data analytics [2 pages, Paul]

With the preparations for exascale computing, simulations with both higher resolution and frequency will be achievable, and the data production rates from these simulations will grow proportionally with the compute power of the machines. The post-processing and analysis of these large datasets will require parallel technologies, and the previously existing technologies are mostly serial.

While parallel technologies exist, their use would significant require changes to the post-processing workflow that currently exists. We choose to take an approach that introduces the fewest changes to the workflow with which scientists are already familiar, namely parallelizing individual steps in the workflow by swapping out the serial scripts with parallel alternatives. This approach will require less user re-education and should improve the adoption rate of the newly developed and adopted technologies. This should also lead to fewer problems encountered during and after deployment of the new technologies.

We have chosen develop our new tools with parallel Python, using MPI4PY for parallelism. Python is excellent for rapid development, reducing the time to deployment. It also allows us to develop tools with a very small code-base, making

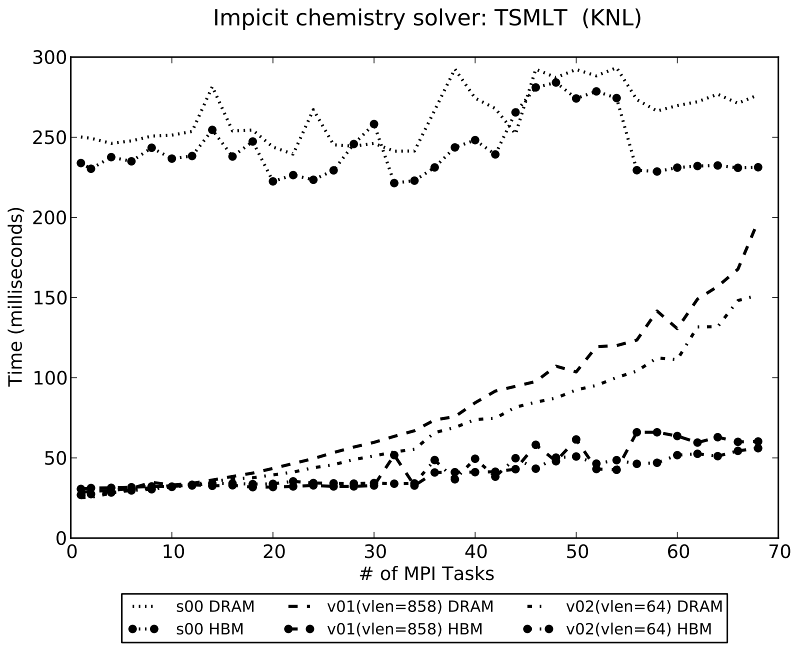

Fig. 16 Weak scaling of several different versions of the implicit chemistry solver on Knights Landing. FIXME: v03 data is missing. Color not allowed in book.

it easier to maintain. Python seamlessly integrates into the existing script-driven CESM workflow, and is cross-platform, reducing the need to address build issues on multiple exascale platforms.

The general CESM post-processing workflow proceeds from the generation of the raw simulation data in the following way. First, since post-processing and analysis tends to be performed on time-series data of only a handful of variables, the data must be transformed from files storing synoptic data (time-slice) to files storing a single variable over a long period of simulated time (time-series). This reduces the amount of data moved when scientists copy datasets for local analysis, and it makes time-series data contiguous within a single file, improving I/O performance with conventional spinning disk storage. The serial tool for performing this transformation is a script written using the NetCDF Operators (NCO), and this serial transformation step has been observed to take as long as the data generation with CESM itself. The parallel Python tool that we have written to replace this NCO script is called the PyReshaper.

The PyReshaper implements a task-parallel approach to speeding up this data transformation; Each output file is defined and written in a single MPI process. In particular, for the time-slice to time-series transformation, this means one output file for each time-series variable in the dataset. This, obviously, has limited scalability, but the immediate benefit is obvious for large datasets which quite often have more than a few dozen time-series variables.

The next step in the workflow is an analysis step, from the time-series files generate with the PYRESHAPER. In this step, temporal and spatial averages are computed for variables in the time-series dataset. These averages, called climatologies, are primarily used to create plots that help scientists quickly assess the correctness of various aspects of the CESM component models. The original script used to perform this task was written using NCO, and the new parallel Python tool that we have written to replace it is called the PYAVERAGER.

The PYAVERAGER implements a parallel approach based on the PYRESHAPER, namely task-parallel with parallelization over the output files. However, since the computation of the averages is a data reduction, it is possible to add an additional level of parallelization over the reduction steps, themselves. Therefore, unlike the PYRESHAPER which dedicates a single MPI process to each output file, the PYAVERAGER dedicates an MPI subcommunicator to each output file, allowing separate MPI processes to be independently responsible for reading and operating on the data. This allows for greater scalability than the PYRESHAPER can achieve, but still limited by the dataset and the kind and number of averages requested.

Both tools were tested in parallel on NCAR's Yellowstone supercomputing platform, located at the NCAR-Wyoming Supercomputing Center (NWSC) in Cheyenne, WY. Two complete CESM datasets were used in these tests: a low-resolution (1-degree, 232 GB) dataset and a high-resolution ($\frac{1}{4}$-degree atmosphere/land, $\frac{1}{10}$-degree ocean/ice, 4.4 TB) dataset, both spanning 10 years of data. At maximum parallelism, the PYRESHAPER reduced the time-slice to time-series transformation step on the low- and high-resolution datasets from 3.7 hours and 23 hours, respectively, to 30 minutes and 2 hours, respectively. That amounts to a 7.6$\times$ and 11.7$\times$ speedup, respectively. The PYAVERAGER reduced the computation of climatologies for the low- and high-resolution datasets from 9 hours and 40 hours, respectively, to 4 minutes and 16 minutes, respectively. That amounts to a 130$\times$ and 150$\times$ speedup, respectively.

Current and future development has been directed toward the application of these parallel Python techniques to more general and sophisticated problems, including the problem of data standardization and publication. During model intercomparison projects, such as the Climate Model Intercomparison Project (CMIP), data generated by many modeling codes and institutions must be collected together for direct comparison with each other. This necessitates data standardization, including conventions for variable naming, units, and additional metadata. The process of standardization involves much that same procedures as those involved in the PYRESHAPER and PYAVERAGER utilities, and these standardizations can even be seen as a generalization of the same tools. This tool, in development at the time of this writing, is called PYCONFORM.

PYCONFORM is a general tool for performing dependent tasks, in a proscribed sequence, on large datasets to produce second datasets of a different format. Such sequence of tasks comprise a directed acyclic graph (DAG), the flow of which describes the flow of data from input dataset to output dataset. Obvious parallelism can be achieved across independent components of the DAG, while greater parallelism can be achieved by further splitting components of the DAG and connecting the components with MPI communication. Further development is planned, and a release is expected before the end of 2016.

Acknowledgements

Need to thank a bunch of people and programs here.

References

[1] A. H. Baker, D. M. Hammerling, M. N Levy, H. Xu, J. M. Dennis, B. E. Eaton, J. Edwards, C. Hannay, S. A. Mickelson, R. B. Neale, D. Nychka, J. Shollenberger, J. Tribbia, M. Vertenstein, and D. Williamson. A new ensemble-based consistency test for the community earth system model. *Geoscientific Model Development*, 8:2829–2840, 2015.

[2] A. H. Baker, Y. Hu, D. M. Hammerling, Y. Tseng, H. Xu, X. Huang, F. O. Bryan, and G. Yang. Evaluating statistical consistency in the ocean model component of the community earth system model (pycect v2.0). *Geoscientific Model Development*, 9:2391–2406, 2016.

[3] Tom Henderson, John Michalakes, Indraneil Gokhale, and Ashish Jha. Numerical weather prediction optimization. In James Reinders and Jim Jeffers, editors, *High Performance Parallelism Pearls Multicore and Many-core Programming Approches Volume two*, chapter 2, pages 7–23. Elsevier, New York, 2015.

[4] Shian-Jiann Lin. A ”vertically lagrangian” finie-volume dynamial core for global models. *Monthly Weather Review*, 132:2293–2397, 2004.

[5] Daniel J. Milroy, Allison H. Baker, Dorit M. Hammerling, John M. Dennis, Sheri A. Mickelson, and Elizabeth R. Jessup. Towards characterizing the variability of statistically consistent community earth system model simulations. In *Procedia Compture Science*, volume 80 of *ICCS 2016. The International Conference on Computational Science*, pages 1589–1600, 2016.

[6] A. J. Simmons and D. M. Burridge. An energy and angular-momentum conserving vertical finite-difference scheme and hybrid vertical coordinates. *Monthly Weather Review*, 109:758–766, 1981.

[7] Mark Taylor, Joseph Tribbia, and Mohamed Iskandarani. The spectral element method for the shallow water equations on the sphere. *Journal of Computational Physics*, 130(1):92–108, 1997.