OpenMP 4 Fortran Modernization of WSM6 for KNL

T.A.J. Ouermi
University of Utah
touermi@sci.utah.edu

Aaron Knoll
University of Utah
knolla@sci.utah.edu

Robert M. Kirby
University of Utah
kirby@sci.utah.edu

Martin Berzins
University of Utah
mb@sci.utah.edu

ABSTRACT

Parallel code portability in the petascale era requires modifying existing codes to support new architectures with large core counts and SIMD vector units. OpenMP is a well established and increasingly supported vehicle for portable parallelization. As architectures mature and compiler OpenMP implementations evolve, best practices for code modernization change as well. In this paper, we examine the impact of newer OpenMP features (in particular OMP SIMD) on the Intel Xeon Phi Knights Landing (KNL) architecture, applied in optimizing loops in the single moment 6-class microphysics module (WSM6) in the US Navy's NEPTUNE code. We find that with functioning OMP SIMD constructs, low thread invocation overhead on KNL and reduced penalty for unaligned access compared to previous architectures, one can leverage OpenMP 4 to achieve reasonable scalability with relatively minor reorganization of a production physics code.

CCS CONCEPTS

•General and reference →Performance; *Validation;* **•Software and its engineering →Multithreading;**

KEYWORDS

parallel, openMP, weather forecasting, overhead, vector parallelism, thread parallelism, Knights Landing

ACM Reference format:
T.A.J. Ouermi, Aaron Knoll, Robert M. Kirby, and Martin Berzins. 2017. OpenMP 4 Fortran Modernization of WSM6 for KNL. In *Proceedings of PEARC17, New Orleans, LA, USA, July 09-13, 2017,* 8 pages.
DOI: 10.1145/3093338.3093387

1 INTRODUCTION

The Weather Research and Forecasting (WRF) Model is an open source numerical weather prediction software used by atmospheric researchers and weather forecasters at operational center and other parties. WRF is designed to help scientists study and better understand weather phenomena. As with many other computational models, significant effort is put into modernizing numerical weather codes for current and future architectures to continue to benefit from Moore's Law [16]. In the case of weather codes, the goals are to improve the accuracy and reduce the time requirements of forecasts.

Traditionally, MPI-level [7] distribution of serial codes has been the primary vehicle for exploiting parallelism in these predominately serial Fortran weather codes. However, in the last decade in particular computational architectures have increased core counts, decreased clock speeds and adopted wide SIMD vector units. The Intel Xeon Phi Knights Landing (KNL) [20] architecture, for example, employs dual 8-lane double precision (DP) floating point units on each of 64 cores running at 1.3 GHz. MPI alone is not suited for this fine granularity; codes must be re-architected to exploit thread and SIMD parallelism.

OpenMP [17] is a compelling model for portable parallelism in that it requires relatively little modification of potentially large, complex codes. However, actual best practices for OpenMP vary widely with the code in question, compiler implementation and underlying architecture. In the past, most effective OpenMP optimizations have used high-level parallel constructs for threading (i.e., mirroring MPI-level parallelism), carefully aligned arrays, and explicit rewrites to eliminate branching. These optimizations are no doubt effective, but require significant modification of existing codes. However, new architectures such as KNL boast lower thread creation times and no longer carry the same penalty for unaligned memory access. OpenMP 4 features such as OMP SIMD promise control over how vectorization is expressed, beyond the autovectorization capabilities of the compiler. Consequently, as OpenMP matures, "naive" approaches may prove almost as effective as wholesale rewrites.

This paper dissects the WSM6 single-moment 6-class [9] microphsyics code in the context of NEPTUNE [5, 6] and examine OpenMP optimization of individual loops, first using synthetic examples and subsequently in WSM6 itself. The experiments suggest several interesting findings – particularly involving thread invocation time on Xeon and KNL and effectiveness of OMP SIMD on code with branching and nested subroutines. Extending these lessons to WSM6, straightforward OMP DO SIMD constructs can deliver over 50x speedup over serial for several loops. Moreover, while not the most effective, low-level OpenMP with OMP DO SIMD can be a valid approach for accelerating serial codes with minimal changes to source.

2 RELATED WORK

Various optimization approaches have been applied to different component of NEPTUNE and other weather prediction systems.

Michalakes et al. [11]. optimized the Weather Model Radiative Transfer Physics by restructuring the code to expose concurrency, vectorization, and locality. In this approach they explicitly reorganized the arrays dimension, and lowered the inner loop size to fit into the vector lane in order to take advantage of the vector units. These optimizations yielded about 3x speedup.

Mielikainen et al. [14] optimize the Purdue-Lin Microphysics Scheme by applying thread level parallelism across the horizontal grid points because there is no communication among them. Furthermore, they collapse the loops into smaller loop sizes, reduce the amount of temporary variables and use SIMD for explicit vectorization. These modifications led to a speed up of 4.7x on the Intel Xeon Phi 7120p. In an similar work Mielikainen et al. [13] improved the Goddard microphysics scheme's, in WRF, performance on the Intel Xeon Phi. In this particular work they focused on exposing vector level parallelism and reducing the memory footprint.
Although This work here focuses on the Intel MIC, there are have been some work on optimizing physics schemes for GPU [12, 15, 18, 19]. The GPU base optimization shows better speed ups than the MIC. For instance Mielikainen et al. [15] using CUDA achieve a speedup of 2 orders of magnitude.

OpenMP is increasingly becoming the standard for shared memory parallelism. It offers a simple high level abstraction for thread and vector parallelism. In order to leverage OpenMP features, different groups investigated the overheads associated with OpenMP directives [1–3, 10]. The overhead is not only dependent of the OpenMP implementation but also of the architecture. LaGrone et al. [10] developed a benchmark for measuring the overhead associated with the tasking model and the synchronization in OpenMP on a 2.27 GHz 8-core Intel Xeon Nethalem E5520 processors. Dimakopoulos et al [4]. studied OpenMP overheads under nested parallelism for different compilers. In both these studies, they extended the EPCC benchmark to include the OpenMP directives of interest. In this work we investigated the overheads on the Intel Knights Landing and the effort necessary to minimize such overhead.

3 OVERVIEW OF NEPTUNE/WRF AND WSM6

The "Navy Environmental Prediction sysTem Utilizing the NUMA corE" (NEPTUNE) is part of the United States Navy's effort to develop the next generation weather prediction system using the Nonhydrostatic Unified Model of the Atmosphere [5, 6]. The Weather Research Forecasting Model infrastructure is composed of multiple modules. The main components are the dynamic solvers and the physics schemes; this work targets the WRF Single-Moment 6-class Microphysics scheme (WSM6).The microphysics scheme is a physical parametrization that simulates processes in the atmosphere that cause precipitation of rain, snow, graupel, water vapor, cloud water, cloud ice. WSM6 is an improvement of WSM5 that introduce graupel particles and other variables to better model precipitation of hydrometeors. The computation in the scheme is organized along a horizontal and a vertical direction. There is no interaction among the horizontal grid points which allows straight forward parallelism cases. Furthermore, the problem size studied, WSM6 supercell test case, fits in the MCDRAM.

4 EXPERIMENTAL SET UP

4.1 Methodology

In order to systematically and rigorously investigate the performance bottlenecks in WSM6 this work used a methodology that consisted of four steps.

(1) Understanding code: This consisted of analysing the loop structures and the data dependencies that exist among the loops.
(2) Identifying bottlenecks: Profiling the code using Intel VTune and wall clock timers to identify the bottlenecks.
(3) Building and testing stand alone experiments base on bottlenecks: we designed standanlone experiments to address the bottlenecks identified in previous steps. These experiments allow us to identify what approach is better suited for a specific bottleneck in WSM6.
(4) Applying findings to WSM6: The findings in step three guide the different optimization decision in WS6 loops.

4.2 Measurement Parameters

This section summarizes experiments conducted to explore various optimization strategies for the WRF WSM6 module on the Intel Knights Landing (KNL). This effort focuses on understanding the KNL and the steps necessary to effectively exploit the resources offered by the KNL architecture. Performance of a given code is evaluated using seven attributes: number of threads, serial time, parallel time, work/thread, overhead, speedup, and efficiency.

- **Overhead**: the overhead associated with thread creation, thread binding, scheduling, etc [10]:

$$Overhead = \frac{n \times T_p - T_s}{n} \tag{1}$$

where n is the number of threads, T_s is the serial time, and T_p is the parallel time.

- **Work/thread**: the average work, in Floating Point Operations Per Second (FLOPS), per thread. The work per thread is calculated by dividing the total amount of work in a loop by the number of threads used.

$$w_{thread} = \frac{\text{work in loop}}{\text{number of threads}} \tag{2}$$

- **Efficiency**:

$$Efficency = \frac{T_s}{n \cdot T_p} \tag{3}$$

where n is the number of threads, T_s is the serial time, and T_p is the parallel time.

4.3 KNL Architecture

The Intel Knights Landing architecture consists of 36 tiles interconnected with a 2D mesh, MCDRAM of 16G High Bandwidth, and 1 socket. It has a clock frequency of 1.3 GHz which is lower than the 2.5 GHz of Haswell. The Knights Landing tile is the basic unit that is replicated across the entire chip. The tile consist of two cores each connected to two Vector Processing Units (VPUs). Both cores share a 1 MB L2 cache. Two AVX-512 vector units process 8 double-precision lanes each; a single core can execute two 512-bit vector multiply-add instructions per clock cycle. The results and experiments presented in this paper use the default thread and processor binding.

5 STANDALONE OPENMP FORTRAN EXPERIMENTS

This section describes standalone experiments designed to verify the functionality of OpenMP, and mimic the behavior of WSM6 in a minimal reproducible fashion. In the following pseudocode, $work(i, j)$ denotes computations similar to

$$a(i, j, 1) = 0.1 * b(i, j, 1) + c(i, j, 1)/d(i, j, 1).$$

The computation is always the same, but different outer dimensions are used to simulate access of multiple arrays. This behavior is similar to the array operations in WSM6.

5.1 Overhead associated with OpenMP on KNL

5.1.1 Overhead Per Thread Minimization. This experiment analyzes different methods that aim to minimize the overhead/thread. It determines the amount of work in FLOPS per thread required to minimize the overhead and maximize the speedup. Synthetic examples similar to the loops in WSM6 are used for this experiment. Code 1, shown below, is used to measure a baseline overhead. Code 2 is used to analyze the overhead/thread of a WSM6-like loop. The variables ie and je are 10592 and 39 respectively. In theory, given "sufficient" work for each thread the performance results from Code 2 should be comparable to Code 1 results.

```
Code 1:
!$OMP PARALLEL
!$OMP DO
DO i=1,100
  work(i)
ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

```
Code2:
 !$OMP PARALLEL
 !$OMP DO
 do j=1, je
   do i=1, ie
     work(i,j)
  end do
end do
!$OMP END DO
!$OMP END PARALLEL
```

n	T_s	T_p	Overhead	w_{thread}	Speedup	Efficiency
2	92.797	46.500	0.22	1000000	1.995	99.78
4	92.865	23.335	0.51	500000	3.98	99.49
8	92.770	12.209	5.00	250000	7.59	94.98
16	92.826	6.608	12.26	125000	14.05	87.79
32	92.944	3.831	24.27	62500	24.26	75.82

Table 1: Performance results from Code 1.

n	T_s	T_p	Overhead	w_{thread}	Speedup	Efficiency
2	27.636	15.848	2.03	1858896	1.74	87.19
4	27.352	8.099	1.26	929448	3.38	84.43
8	27.439	4.108	0.68	464724	6.68	83.49
16	27.422	2.586	0.87	232362	10.60	66.27
32	27.507	1.780	0.92	116181	15.45	48.29

Table 2: Modified Code 2 with !$OMP DO placed outside j loop.

The results from Table 1 show that the average overhead/thread is less than 1 microsecond. By increasing the number of threads, the number of FLOPS per thread decreases. This decrease causes the overhead per thread to increase. The minimal overhead/thread, 0.1 microsecond, is observed at about 1 million FLOPS per thread. However, with 0.03 million FLOPS per thread, the measured overhead remains below 1 microsecond.
The results from Table 2 show higher overheads and lower speedups compared to Table 1 results. Code 2 is structured differently compared to Code 1. The compuatation in *work(i)* is done with a 1D array while the computation in *work(i,j)* is done with a 2D array. Futhermore the dimension of the arrays are different. These differences explain different observed overheads and speedups.
When the !$OMP PARALLEL and !$OMP DO are moved to the i loop or !$OMP PARALLEL at the j loop and !$OMP DO at the i loop, larger overheads and lower speedups occur, compared to the results from Tables 1 and 2

5.1.2 Keeping Threads Active/Alive. Keeping threads active/alive during computation reduces thread creation and cancellation overheads. !$OMP PARALLEL is the directive that creates the pool of threads (fork), and !$OMP END PARALLEL cancels the created threads (join). Thus, creating threads at the beginning of a computation and canceling them at the end should reduce the overhead associated with the creation and cancellation of threads. Furthermore, the OpenMP environment variable KMP_BLOCKTIME can be used to keep threads alive for a certain Time. This experiment compares a single parallel block performance against multiple parallel blocks. One would expect Code 4 to outperform Code 3. Because Code 4 is constructed with and single parallel block, it does not incur the thread creation overhead caused by the multiple !$OMP PARALLEL blocks.

```
Code 3:
!$OMP PARALLEL
!$OMP DO
  do j=1, je
    do i=1, ie
      3 x work(i,j)
    end do
   end do
 !$OMP END DO
 !$OMP END PARALLEL

  !$OMP PARALLEL
  !$OMP DO
  do j=1, je
    do i=1, ie
      3 x work(i,j)
    end do
  end do
  !$OMP END DO
  !$OMP END PARALLEL

  !$OMP PARALLEL
  !$OMP DO
  do j=1, je
    do i=1, ie
     3 x work(i,j)
    end do
  end do
```

```
Code 4:
!$OMP PARALLEL
!$OMP DO
  do j=1, je
    do i=1, ie
      3 x work(i,j)
    end do
  end do
  !$OMP END DO

  !$OMP DO
  do j=1, je
    do i=1, ie
      3 x work(i,j)
    end do
  end do
  !$OMP END DO

  !$OMP DO
  do j=1, je
    do i=1, ie
      3 x work(i,j)
    end do
  end do
  !$OMP END DO
  !$OMP END PARALLEL
```

```
    !$OMP END DO
    !$OMP END PARALLEL
```

n	T_s	T_p	Overhead	Speedup	Efficiency
2	81.597	47.119	6.32	1.73	86.58
4	81.398	24.080	3.73	3.38	84.50
8	81.222	12.360	2.20	6.57	82.14
16	81.357	7.479	2.39	10.87	67.98
32	81.755	5.150	2.59	15.87	49.60

Table 3: Multiple parallel blocks with KMP_BLOCKTIME in Code 3.

n	T_s	T_p	Overhead	Speedup	Efficiency
2	81.597	46.300	5.50	1.76	88.12
4	81.398	23.560	3.21	3.45	86.37
8	81.222	12.021	1.87	6.75	84.46
16	81.930	7.188	2.07	11.39	71.24
32	81.755	5.028	2.47	16.26	50.81

Table 4: Single parallel block with KMP_BLOCKTIME in Code 4.

Table 3 and Table 4 show performance results for multiple parallel blocks and a single parallel block, respectively. The single parallel block from Code 4 performs slightly better than the multiple blocks case from Code 3, which supports the initial assumptions.

5.1.3 *OpenMP Versus Pthreads Overhead.*

This experiment analyzes the overhead associated with thread creation and context switches in the OpenMP and Pthreads libraries. In order to establish a fair comparison both libraries, the synthetic experiment has been done in C, because Pthreads does not have an equivalent in Fortran.

	Pthreads		OpenMP	
n	Thread creation	Context	Thread creation	Context
2	199	0.631	14311	6.017
4	183	0.736	8047	3.336
8	121	1.182	4209	1.375
16	107	1.067	4115	1.046
32	102	1.041	1654	0.797
64	99	1.035	1417	0.943
128	120	1.27	653	1.579

Table 5: Thread creation and context switch overhead measurements with Pthreads and OpenMP.

Table 5 shows the thread creation overhead and context switches measurements for Pthreads and OpenMP. These results indicate that OpenMP has significantly higher thread creation overhead compared to Pthreads. the context switch measurements observed in OpenMP are slightly but not significantly higher than the ones in Pthreads but not significant.
These experiments show that the use of a single parallel block coupled with the environment variable KMP_BLOCKTIME contributes to reducing the overheads and increasing the speedups slightly.

5.1.4 *KNL vs Haswell overhead.*

Thread overhead is dependent on the implementation of OpenMP and the architecture used. Here, performance of Code 1 on KNL and Haswell are compared.

n	T_s	T_p	Overhead	w_{thread}	Speedup	Efficiency
2	67.825	34.225	0.31	1000000	1.981	99.08
4	67.77	17.263	0.32	500000	3.925	98.14
8	67.729	9.079	0.61	250000	7.459	93.25
16	68.099	5.015	0.76	125000	13.579	84.86
32	67.973	2.861	0.73	62500	23.758	74.25

Table 6: Performance results from Code 1 on Haswell.

The results from Tables 1 and 6 indicate that the KNL have a lower overhead than Haswell. The average overhead on KNL is about 0.51 whereas the overhead on Haswell is about 0.61. In addition, the speedups observed on KNL are greater than the ones on Haswell.

5.2 Thread Scalability

5.2.1 *Base case.*

The previous section analyzed examples of loops that do not exhibit significant complexity. This section focuses on understanding the performance impact of function calls. The example examined here is a loop with nested functions calls.

This experiment analyzes a base case that is used as a reference. Code 5 measures, how its performance scales with the number of threads before transformation.

```
Code 5 : WSM6 loop with conditionals and function calls
    do k = kte, kts, -1
      do i = its, ite
        ...
        if(t(i,k).gt.t0c) then
            ...
          work2(i,k) = venfac(p(i,k),t(i,k),den(i,k))
          if(qrs(i,k,2).gt.0.) then
           ...
            psmlt(i,k) = xka(t(i,k),den(i,k))
            ...
          endif
          if(qrs(i,k,3).gt.0.) then
            ...
            pgmlt(i,k) = xka(t(i,k),den(i,k))
            ...
          endif
        endif
      enddo
    enddo
```

A close analysis of Code 5 shows two function calls: *xka* and *venfac*. These functions are implemented by calling two other functions, *viscos* and *diffus*. Furthermore, all the functions call intrinsic math functions such as *sqrt*, for which most current compilers now emit vector instructions.

n	T_s	T_p	Overhead	Speedup	Efficiency
2	2109.055	944.316	-110.21	2.23	111.67
4	2107.635	499.088	-27.82	4.23	105.57
8	2109.226	262.901	-0.75	8.02	100.27
16	2109.020	234.1	102.28	9.01	56.30
32	2110.18	231.294	165.35	9.12	28.51
40	2109.158	218.183	165.45	9.67	24.16

Table 7: Performance results from Code 5.

Table 7 shows that Code 5 scales up to 8 threads. After 8 threads the overhead increases drastically and the speedup plateaus at about 9x.

n	T_s	T_p	Overhead	Speedup	Efficiency
2	90.567	57.241	11.95	1.58	79.11
4	90.427	29.06	6.45	3.11	77.79
8	90.480	14.7601	3.45	6.13	76.62
16	90.581	8.721	3.06	10.39	64.92
32	90.631	5.882	3.05	15.41	48.15
40	90.973	3.064	0.79	29.69	74.22

Table 8: Code 5 without function calls and conditionals.

As mentioned before Code 5 has nested functions and conditionals. Such complexities may cause performance limitations for threading and vectorization.
Table 8 shows performance results from Code 5 with all function calls and conditionals removed. By removing the function calls, the amount of computation in the loop is significantly reduced. This reduction resulted in the serial time in Table 8 being much smaller than that in Table 7. Table 8 has significantly higher speedups and lower overheads than Table 7. These results indicate that the conditionals and the function calls are responsible for the performance limitations observed.

5.2.2 *Function Calls Performance Analysis.*

As mentioned above, Code 5 has nested function calls. This experiment compares the performance of a modified version of Code 5 against Code 6. In the modified version of Code 5, the conditionals have been removed but the function calls left intact. In Code 6 the function calls are replaced by some code that performs the same task as the functions.

```
Code 6 : WSM6 complex Code with no function calls.
   !$OMP PARALLEL DEFAULT(shared) PRIVATE(i, k)
   !$OMP DO
   do k = kte, kts, -1
     do i = its, ite
       ...
       !!--work2(i,k) = venfac(p(i,k),t(i,k),den(i,k))
       temp0 =  1.496e-6 * (t(i,k)*sqrt(t(i,k))) /  &
               (t(i,k)+120)/den(i, k)
       temp1 =  8.794e-5 * exp(log(p(i,k))* (1.81)) / t(i,k)
       work2(i,k) = exp(log((temp0/temp1))* ((.3333333)))  &
              /sqrt(temp0)*sqrt(sqrt(den0/den(i,k)))

       !!-- xka(t(i, k), den(i, k))
       temp3 = 1.414e3*1.496e-6 * (t(i,k)*sqrt(t(i,k)))/
               (t(i,k)+120)/den(i, k)*den(i,k)
       ...
   enddo
 enddo
 !$OMP END DO
 !$OMP END PARALLEL
```

n	T_s	T_p	Overhead	Speedup	Efficiency
2	1909.236	594.708	-359.910	3.21	160.51
4	1909.292	297.209	-180.114	6.42	160.60
8	1909.105	149.470	-89.17	12.77	159.65
16	1909.222	89.801	-29.52	21.26	132.88
32	1910.795	60.320	0.61	31.68	98.99
40	1910.146	30.584	-17.17	62.45	156.15

Table 9: Performance results from Code 6.

The modified version of Code 5 yielded a maximum speedup of about 3x whereas Code 6 yielded a maximum speedup of 62x. Table 9 report the results from Code 6.

5.2.3 *Subroutine Calls Performance Analysis.*

This experiment measures the performance impact of subroutine calls. Code 7 below contains a subroutine call and few conditionals.

```
Code 7
 !$OMP PARALLEL DEFAULT(shared) PRIVATE(i,j,m,thread_id)
 do j=1, je
   thread_id = OMP_GET_THREAD_NUM()

   !$OMP DO SIMD
   do i=1, ie
     do m=1, M_LOOPS
     #if OMPTEST_SUBROUTINE
     call do_work(i,j)
     #else
    3 x work(i,j)
     if (b(i,j,1) .gt. 0.0) then
      3 x work()
     endif
     #endif
     enddo
   enddo
   !$OMP END DO SIMD NOWAIT
   end do
   !$OMP END PARALLEL

 subroutine do_work(i,j)
 !$OMP DECLARE SIMD(do_work)
 integer :: i,j
3 x work()
 if (b(i,j,1) .gt. 0.0) then
  3 x work(i,j)
 endif
 end subroutine do_work
```

Table 10 reports the following experimental cases:

- case 1 represents results from Code 7 with OMP PARALLEL + OMP DO SIMD;
- case 2 represents results from Code 7 with OMP PARALLEL + OMP DO SIMD and subroutine; and
- case 3 represents results from Code 7 with OMP PARALLEL + OMP DO SIMD, subroutine and DECLARE SIMD on functions.

case	n	T_s	T_p	Speedup	Efficiency
case 1	64	69	1.46	47.26	73.84
case 2	64	69	1.45	62.09	74.35
case 3	64	90	1.48	60.81	95.02

Table 10: Code 7 results.

Table 10 shows the performance result from Code 7. The DECLARE SIMD use in this experiment does not have a significant performance impact. Table 10 cases 2 and 3 report additional variations that were tested as given by their captions. These results show significant speedups. This indicates that a single level (no nesting) subroutine and and conditionals are not a performance bottleneck and various combination of !$OMP PARALLEL and !$OMP SIMD yield significant speedups.

5.2.4 *Nested Conditionals.*

This experiment focuses on studying the performance impact of conditionals. It compares the performance of Code 9 against a modified versions of Code 6. The modified version of Code 6 include nested conditionals.

```
Code 9 : WSM6 loop with masking and no function calls.
compute bool_val1, bool_val2, and bool_val3

!$OMP PARALLEL DEFAULT(shared) PRIVATE(i, k)
!$OMP DO
 do k = kte, kts, -1
   do i = its, ite
       ..
       !!--work2(i,k) = venfac(p(i,k),t(i,k),den(i,k))
        temp0 =  1.496e-6 * (t(i,k)*sqrt(t(i,k))) / &
               (t(i,k)+120)/den(i, k)
        temp1 =  8.794e-5*exp(log(p(i,k))*(1.81))/t(i,k)
       work2(i,k) = exp(log((temp0/temp1))* ((.3333333))) &
          /sqrt(temp0)*sqrt(sqrt(den0/den(i,k)))
       ...
       compute result1
         ...
       !!-- xka(t(i, k), den(i, k))
        temp3 = 1.414e3*1.496e-6 * (t(i,k)*sqrt(t(i,k)))/ &
          (t(i,k)+120)/den(i, k) &
          den(i,k)
       ...
       compute result2
       final_result = (result1*bool_val1 &
          + result2*bool_val2)*bool_val0
   enddo
 enddo
 !$OMP END DO
 !$OMP END PARALLEL
```

n	T_s	T_p	Overhead	Speedup	Efficiency
2	2102.487	691.904	-359.34	3.04	151.93
4	2099.805	344.800	-180.15	6.09	152.24
8	2099.784	172.063	-90.41	12.20	154.25
16	2106.650	104.066	-27.60	20.24	126.52
32	2112.524	69.479	3.46	30.40	95.02
40	2100.923	35.199	-17.32	59.69	141.22

Table 11: Performance results from a Code 9.

The modified version of Code 6 obtains a maximum speedup of about 8x. Using SIMD decrease the speedup to about 3x. SIMD fails to vectorize because of the nested conditionals. Moving the conditionals outside the loops as shown in Code 9 to address the bottleneck observed. This transformation yields significant speedups and low overheads as shown in Table 11.
These experiments indicate that nested conditionals hurt performance. Eliminating branching yields significant improvements. This approach can be used in many of the WSM6 codes that exhibit similar patterns.

5.3 Vectorization

5.3.1 OMP SIMD. This section analyzes the performance impact of SIMD. In this experiment a simple compute-only code is considered. It compares the performance of various parallel versions and Code 8 against the serial version. Furthermore, the thread binding is done manually.

```
Code 8 : WSM6 loop with masking and no function calls
!$OMP PARALLEL DEFAULT(none)
!$OMP SHARED(a, b, c, d, je, ie,num_tasks)
!$OMP PRIVATE(i,j,m,its,ite,Thread_id)
thread_id = omp_get_thread_num()
its = 1 + thread_id * num_tasks * VLEN
ite = min(its + num_tasks * VLEN - 1, ie)
do j=1, je
!$OMP SIMD
do i=its, ite
  do m=1, 10
    3 x work(i,j)
  enddo
  if (b(i,j,1) .gt. 0.0) then
    3 x work(i,j)
  endif
enddo
!$OMP END SIMD
!$OMP END PARALLEL
```

Table 12 reports the following experimental cases:

- case 1 represents results from Code 8 with OMP DO placed right before the i loop;
- case 2 represents results from Code 8 with only manual implementation of thread binding;
- case 3 represents results from Code 8 withwith OMP DO SIMD at the i loop; and
- case 4 represents results from Code 8 SIMD and manual implementation of thread binding.

case	n	T_s	T_p	Speedup	Efficiency
case 1	64	69	2.88	24	37.43
case 2	64	69	0.63	109	171.13
case 3	64	69	0.614	112	175.59
case 4	64	69	0.614	112	175.59

Table 12: Code 8 results.

Table 12 shows the speedups when the different directives are placed right before the i loop. OMP PARALLEL + OMP SIMD yields the highest speedup among the different experiments.

6 MODERNIZATION OF WSM6

6.1 Code Overview

The WSM6 supercell test case of WSM6 consists of 27 loops around 10K (i) rows, with three subroutines (slope_wsm6, nislfv_rain_plm, nislfv_rain_plm6) [8]. The last two subroutines contain non-trivial control flow (cycle/goto statements). The other loops are generally memory intensive, with significant branching.
Applying the findings of the standalone tests, WSM6 was modified with OpenMP directives as follows:

- OpenMP initialization code in init_microphysics() in mod_microphysics.f90;
- Consistent use of OMP PARALLEL and OMP DO SIMD as presented in case 3 of Table 12;
- Minor code modification to remove nested conditionals and function calls as demonstrated in Code 9;
- Use of OMP PARALLEL sections around multiple loops as shown in Code 3 to reduce thread invocation overhead;
- Elimination of false sharing and specification of PRIVATE variables ;
- Merging smaller loops involving the same arrays, to mitigate thrashing; and

- Using C preprocessor macros to enable and measure runtime of parallel, serial or both implementations.

The decision to first pursue OMP DO SIMD, despite worse performance than OMP SIMD with manual indices from Table 12, was due to the presence of many temporary arrays in between loops and dependency-heavy subroutines in WSM6. Moreover, the aim is to first explores what naive "low-level" OpenMP parallelization could deliver with miminal reorganization of the code.

6.2 WSM6 Results

Tests were conducted on a 4-socket (Haswell) Intel Xeon E7-8890 v3 with 3 TB RAM, and Intel Xeon Phi 7210 ("Knights Landing", or KNL) with 16 GB MCDRAM and 96 GB DRAM. The compiler was Intel Parallel Studio 2016, update 3 (build 67), due to issues with Parallel Studio 2017 in other modules within NEPTUNE.

Overall scalability. Results up to all cores (64 cores on KNL, 72 on Haswell) on these respective systems are shown in Table 13, for different values of OMP_NUM_THREADS. We see limited benefit from hyperthreading on either platform. Though not shown in the table, we found that 18 cores of KNL performed 2.12x better than a single-socket equivalent Haswell (with OMP_NUM_THREADS=18). Moreover, using the default maximum number of threads (144 on Haswell, 256 on KNL), KNL performs roughly 2x faster than Haswell core-for-core. This suggests better scalability on KNL than on Haswell. We note, however, that we used default thread affinity settings (i.e., KMP_AFFINITY). Our Brickland-EX Haswell system has a non-conventional memory architecture supporting up to 6 TB RAM, generally exhibiting higher inter-socket latencies than comparable Xeon workstations, which could affect performance. Further work may be required to scale specifically on this platform.

	Haswell		KNL		KNL vs HSW
n	T_p	Speedup	T_p	Speedup	KNL vs HSW
1	0.46	1	1.77	1	0.26
4	0.116	4	0.222	8	0.52
16	0.068	6.9	0.067	26	1
32	0.067	7.9	0.04	44	1.7
64	0.064	18	0.031	57	2.1
128	0.06	19	0.035	50	1.7
256	0.19	6	0.037	47	5

Table 13: Scalability and speedup (over serial) on 72-core Haswell-EX and 64-core KNL with different values of OMP_NUM_THREADS.

6.3 WSM6 Discussion

Scalability challenges. Though these results show good performance on KNL, the current implementation does not scale well beyond 36 cores (2-socket equivalent) on Xeon. This is perhaps a result of slightly higher thread invocation time on Xeon, but more likely due to the higher clock speed of that architecture and worse "base" speedup of threads compared to serial. Though the use of multiple parallel sections scales well on KNL, these standalone experiments with thread overhead suggest that fewer parallel sections, and use of manual indexing (i.e. OMP PARALLEL and OMP SIMD instead of OMP DO SIMD directives) are necessary for better Xeon performance.

	KNL		Haswell	
Loop	T_s	T_p	T_s	T_p
Init loops	2.88×10^{-3}	–	2.92×10^{-3}	–
loop 1	16.8	1.19	6.76	1.58
loop 2	122	1.37	15.6	3.45
loop 5	46.8	1.28	15.9	1.58
loop 7	17.9	1.22	6.56	1.57
slope_wsm6	98.6	1.81	41.5	4.76
loop 8	19.1	1.33	7.77	2.10
rain_plm	260	–	39.0	–
rain_plm	220	–	62	–
loop 9-11	10.3	1.49	11.7	3.12
slope_wsm6	60.7	1.48	13.0	4.51
loop 12-14	176	2.37	36.1	1.86
loop 15-17	2.96	0.859	2.53	0.598
loop 18-19	102	2.26	13.1	2.36
slope_wsm6	59.7	1.77	12.5	4.42
loop 20-21	524	5.32	76.4	5.92
loop 22	246	4.32	119	9.91
loop 23	193	1.95	32.6	6.02
loop 24-26	156	2.99	26.8	3.65
loop 27	4.52	5.85	5.61	6.98
wsm6 total	1860	38.9	440	64.3

Table 14: Wall clock time measurements of individual WSM6 loops, in milliseconds, on 72-core Haswell-EX and 64-core KNL, using all available hardware threads.

	KNL		Haswell	
Loop	speedup	Efficiency	speedup	Efficiency
Init loops	–	–	–	–
loop 1	14.06	21.00	4.91	5.94
loop 2	89.10	139.14	4.53	6.28
loop 5	36.44	56.15	10.07	13.97
loop 7	14.67	22.92	4.16	5.80
slope_wsm6	54.56	85.12	8.71	12.08
loop 8	14.44	22.43	3.70	5.13
rain_plm	–	–	–	–
rain_plm	–	–	–	–
loop 9-11	6.90	10.80	3.73	5.20
slope_wsm6	40.87	64.08	2.95	4.00
loop 12-14	74.41	116.03	19.36	26.95
loop 15-17	3.45	5.38	4.22	5.87
loop 18-19	45.11	70.51	5.56	7.70
slope_wsm6	33.69	52.70	2.83	3.93
loop 20-21	98.48	153.90	12.93	17.92
loop 22	57.13	88.97	12.01	16.68
loop 23	99.53	154.64	5.42	7.52
loop 24-26	52.26	81.52	7.31	10.19
loop 27	0.77	1.20	0.80	1.11
wsm6 total	47.81	74.71	6.91	9.50

Table 15: Speedup over serial from Table 14.

Remaining bottlenecks. Non-parallelizable and poorly parallelizable subroutines in wsm6 and mod_microphysics remain a bottleneck. Complicated subroutines with dependencies such as nislfv_rain_plm and nislfv_rain_plm6 require extensive rewrites to achieve speedup; currently only 2x speedup is achieved for the former. More significantly, horizontal-to-vertical memory copies of arrays in both WSM6 and mod_microphysics in Neptune are difficult to parallelize, achieving at best 2x speedup. These require further investigation. In addition to bottlenecks originating from loops with dependencies, the microphysics code as a whole remains bottlenecked by horizontal-to-vertical copying and integration of arrays. To address this, one should consider interleaving these copy statements with parallel computation, and re-evaluating how data are warehoused in the calling code.With bottlenecks, the entire microphysics module achieves only a modest 3x improvement over serial on KNL, and 2x on Xeon. Ultimately, one would like to restructure all WSM6 directives, moving from OMP DO SIMD to chunks with OMP SIMD and a single high-level OMP PARALLEL section, similar to the approach of Michalakes et al. [11]. This will require parallelizing all remaining sections and eliminating copies of full arrays within WSM6.

Flat vs. Cache Mode on KNL. The memory modes of the Xeon Phi KNL architecture are of significant interest in many code modernization efforts. In "flat mode", the 16 GB MCDRAM are treated as main memory by the OS; in "cache mode" the MCDRAM serve as a cache for larger pool of DRAM (96 GB on the workstation). In principle KNL hosts deployed in cache mode incur higher cache miss costs, as memory are pulled from DRAM. WSM6 results showed negligible difference between flat and cache modes (in fact, an unexpected 1% advantage for cache mode, which is within the 5% margin of error between individual timesteps of the microphysics code). This merits further investigation, but for the current work we conclude the difference between flat and cache modes are not major factors in the runtime of WSM6.

Summary. Overall speedups achieved compared to serial are convincing: 57x over serial on KNL suggests 5.6% of peak (1024x, 64 cores x 16 SIMD lanes). Although this corresponds only to easily parallelizable loops within WSM6, it encompasses non-trivial code with branching, subroutines and incoherent memory access. In all, the WSM6 work is encouraging in that significant speedup was possible with relatively small changes to code – exactly what is desired in a code portability effort.

7 CONCLUSION AND FUTURE WORK

We have examined the impact of OpenMP directives on a Fortran-based WSM6 microphysics code in WRF. In standalone experiments, we measured the cost of thread overhead and tested the effectiveness of various directives with and without OMP SIMD. These suggest that while greater scalablity may be possible with high-level OpenMP constructs, parallelization of dependency-free code sections is possible with few modifications to the original code. Moreover, we have found cases in which straightforward low-level OpenMP methodologies may work, delivering satisfactory 50x–100x speedups over serial. The fact that modern compilers can emit reasonably efficient threaded and SIMD instructions from complex code with branching, subroutines and unaligned arrays suggests the OpenMP methodology holds promise.
Future work in this effort will pursue a more traditional high-level OpenMP approach in the spirit of Michalakes et al. [11], and more explicitly compare OMP SIMD to branch removal with autovectorization. Hybrid performance with MPI would be of interest as well; strong scaling would reduce the effective size of arrays and pose new challenges for thread and SIMD-level parallelism. Lastly, a goal is to apply these methodologies to other weather physics modules in NEPTUNE, such as GFS operational physics.

ACKNOWLEDGMENTS

This work is supported by DOD PETTT contract PP-CWO-KY07-001-P3, and by the Intel Parallel Computing Centers program. The authors would like to thank Alex Reinecke and Kevin Viner at Navy Research Laboratory, John Michalakes at UCAR, and Rajiv Bendale at Engility Corporation for their help and insights.

REFERENCES

[1] J. M. Bull. 1999. Measuring Synchronisation and Scheduling Overheads in OpenMP. *Proceeedings of First European Workshop on OpenMP* (Spet. 1999), 99–105.
[2] J. M. Bull and D. O'Neil. 2001. A Microbenchmark Suite for OpenMP 2. *Proceeedings of Third European Workshop on OpenMP (EWOMP'01)* (Spet. 2001).
[3] J. Mark Bull, Fiona Reid, and Nicola McDonnell. 2012. A Microbenchmark Suite for OpenMP Tasks. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World (IWOMP'12)*. Springer-Verlag, Berlin, Heidelberg, 271–274. DOI: http://dx.doi.org/10.1007/978-3-642-30961-8_24
[4] Vassilios V. Dimakopoulos, Panagiotis E. Hadjidoukas, and Giorgos Ch. Philos. 2008. . Springer Berlin Heidelberg, Berlin, Heidelberg, 1–12. DOI: http://dx.doi.org/10.1007/978-3-540-79561-2_1
[5] James D. Doyle. 2017. A Next Generation Atmospheric Prediction System for the Navy. (January 2017). https://ams.confex.com/ams/97Annual/webprogram/Paper304323.html.
[6] James D. Doyle, S. Gabersek M. Martini D. D. Flagg J. Michalakes D. R. Ryglicki P. A. Reinecke, K. C. Viner, and F. X. Giraldo. 2017. Next Generation NWP Using a Spectral Element Dynamical Core. (January 2017).
[7] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard*. Technical Report. Knoxville, TN, USA.
[8] Hann-Ming Henry Juang and Song-You Hong. 2009. Forward Semi-Lagrangian Advection with Mass Conservation and Positive Definiteness for Falling Hydrometeors. *Monthly Weather Review* 138, 5 (Septempber 2009), 1778–1791. DOI: http://dx.doi.org/10.1175/2009MWR3109.1
[9] Song-You Hong and Jeong-Ock jade Lim. 2006. The WRF Single-Moment 6-Class Microphysics Scheme (WSM6). *Journal of the Korean Meteorological Society* 42, 2 (April 2006), 129–151.
[10] James LaGrone, Ayodunni Aribuki, and Barbara Chapman. 2011. A Set of Microbenchmarks for Measuring OpenMP Task Overheads. *PDPTA* 2 (November 2011), 594–600.
[11] John Michalakes, Michael J. Iacono, and Elizabeth R. Jessup. 2016. Optimizing Weather Model Radiative Transfer Physics for Intelfis Many Integrated Core (MIC) Architecture. *Parallel Processing Letters* 27, 04 (2016), 1650019. DOI: http://dx.doi.org/10.1142/S0129626416500195
[12] J. Michalakes and M. Vachharajani. 2008. GPU acceleration of numerical weather prediction. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–7. DOI: http://dx.doi.org/10.1109/IPDPS.2008.4536351
[13] Jarno Mielikainen, Bromin Huang, and Hung-Lung Allen Huang. 2014. Intel Xeon Phi accelerated Weather Research and Forecasting (WRF) Goddard microphysics scheme. *Geoscientific Model Development Discussions* 7, 6 (December 2014), 8941–8973. DOI: http://dx.doi.org/10.5194/gmdd-7-8941-2014
[14] J. Mielikainen, B. Huang, and H. L. A. Huang. 2016. Optimizing Purdue-Lin Microphysics Scheme for Intel Xeon Phi Coprocessor. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 9, 1 (Jan 2016), 425–438. DOI: http://dx.doi.org/10.1109/JSTARS.2015.2496583
[15] J. Mielikainen, B. Huang, H. L. A. Huang, and M. D. Goldberg. 2012. Improved GPU/CUDA Based Parallel Weather and Research Forecast (WRF) Single Moment 5-Class (WSM5) Cloud Microphysics. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing* 5, 4 (Aug 2012), 1256–1265. DOI: http://dx.doi.org/10.1109/JSTARS.2012.2188780
[16] G. E. Moore. 2006. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter* 11, 5 (Sept 2006), 33–35. DOI: http://dx.doi.org/10.1109/N-SSC.2006.4785860
[17] OpenMP Architecture Review Board. 2013. OpenMP Application Program Interface Version 4.0. (2013). http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf
[18] Erik Price, Jarno Mielikainen, Bormin Huang, HungLung A. Huang, and Tsengdar Lee. 2013. GPU acceleration experience with RRTMG long wave radiation model. (2013). DOI: http://dx.doi.org/10.1117/12.2031450
[19] Greg Ruetsch, Everett Phillips, and Massimiliano Fatica. 2010. GPU Acceleration Of The Long-wave Rapid Radiative Transfer Model in WRF Using CUDA Fortran. (2010).
[20] James Jeffers James Reinders Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming*.