Improving Scalability and Accelerating Petascale Turbulence Simulations Using OpenMP

M. P. Clay[1], D. Buaria[2], P. K. Yeung[1]
E-mail: mclay6@gatech.edu

[1]Georgia Institute of Technology
[2]Max Planck Institute for Dynamics and Self-Organization

Supported by NSF and SMART (DoD) Program
NCSA Blue Waters (NSF PRAC) and OLCF Titan (INCITE)

OpenMP Developers Conference
Stony Brook University, NY, September 18-20, 2017

Outline

Outline

Introduction and Background

Who are we, and how do we operate?

- High performance computing (HPC) end users, i.e., domain scientists.
- Learn how to use OpenMP by attending workshops, reading books and the standard, and bugging consultants and compiler developers.

What are we interested in computing? (more on the next two slides)

- Turbulent fluid flows and turbulent mixing processes.
- Simplified flow geometries to focus on the fundamental issues.

What are our objectives when using OpenMP?

- Homogeneous computing: can we add OpenMP to improve scalability of an MPI code? How should we use the threads?
- Heterogeneous computing: can we accelerate the computations while still maintaining good scalability to production problem sizes?

Turbulence as a Grand Challenge in HPC

Turbulence is ubiquitous in nature and engineering

- Multiscale problem: unsteady fluctuations in 3-D space and time
- Wide range of scales requires massive grids: 8192^3 (Yeung *et al.*) and 12288^3 (Ishihara *et al.*) current state of the art for the velocity field.

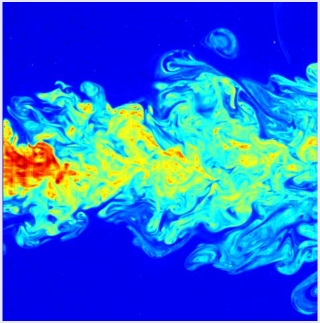
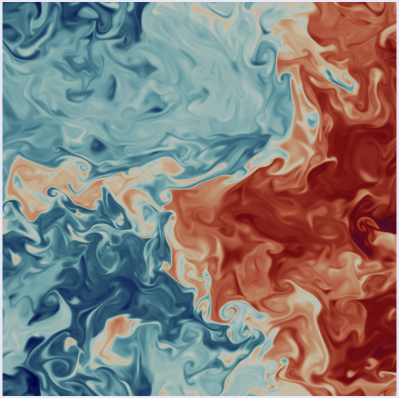


Figure: Images (from wikipedia) of (left) turbulent flow around a submarine and (right) a turbulent jet.

Challenges Facing Simulations of Turbulent Mixing

When the scalar is weakly-diffusive (e.g., salinity in the ocean), resolution requirements for scalar are stricter than the velocity field.



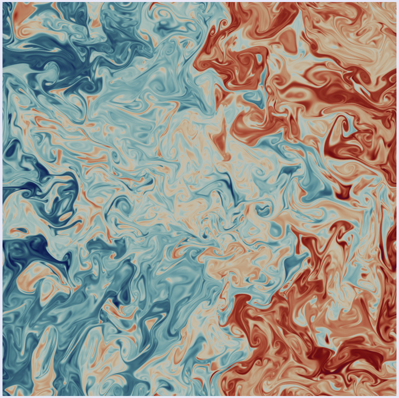


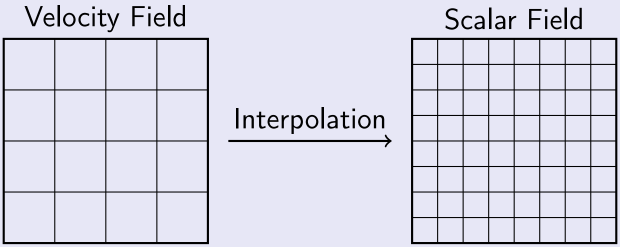
Figure: Scalar fluctuations for (left) a low-diffusivity scalar and (right) a scalar with even lower diffusivity in same (statistically) turbulence.

A Dual-Grid Dual-Scheme Approach

- Velocity field: coarse grid, N-S equations, Fourier pseudo-spectral scheme
- Scalar fluctuations (main interest) on finer grid (Gotoh *et al.*, JCP 2012)

$$\partial\theta/\partial t + \boldsymbol{u}\cdot\nabla\theta = D\nabla^2\theta - \boldsymbol{u}\cdot\nabla\langle\Theta\rangle$$

  - Derivatives via eighth-order combined compact finite differences (CCD)
  - Interpolate velocity from coarse grid to fine grid for advection terms

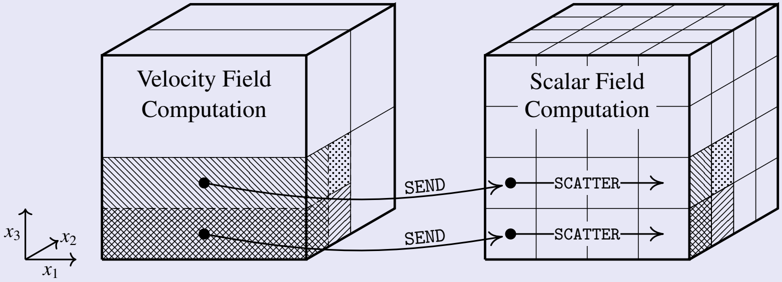


- For our simulations, scalar grid is finer than the velocity grid by a factor of 8 in each direction $\Longrightarrow$ computational cost dominated by scalar

Parallel Implementation for Weakly-Diffusive Scalars

Disjoint groups of processors for the two fields (Clay *et al.* CPC 2017)

- To form advective terms, send well-resolved velocity field to scalar communicator, and perform tricubic interpolation
- Overlap inter-communicator transfer with computations for scalar



Our focus here is on how OpenMP is used for the scalar field computations appearing on the right (the larger computation).

Outline

CCD Scheme and Opportunities to Improve Scalability

Application of the CCD scheme is the most expensive part of the code

- Extract a kernel to focus on the scalability and performance of CCD

Scheme is implicit: all points along grid line coupled

- Parallel algorithm (Nihei *et al.* 2003) to solve system w/o transposes

Op.	Operation Summary
A	Fill ghost layers for scalar field with `SEND` and `RECV` operations
B	Form right-hand-side of linear system and obtain solution
C	Pack and distribute data for reduced system with `MPI_ALLTOALL`
D	Unpack data and solve reduced linear system
E	Pack and distribute data for final solution with `MPI_ALLTOALL`
F	Unpack data and finalize solution of CCD linear system

- Operations for three coordinate directions are independent
  - Try to overlap communication with computation
- Reduce communication requirements (e.g., number of ghost layers) by using OpenMP threads in favor of MPI processes

Dedicated Communication Threads with OpenMP

Initial attempts to improve scalability, and potential problems:

- Only loop-level OpenMP: many threads idle during communication
- Non-blocking MPI: machine-dependent performance (Hager 2011)

Explicitly overlap communication and computation by dedicating thread(s) to each respective operation (Rabenseifner 2003)

- Still operate in `MPI_THREAD_FUNNELED` mode: adjust the number of communication threads by changing the number of MPI processes

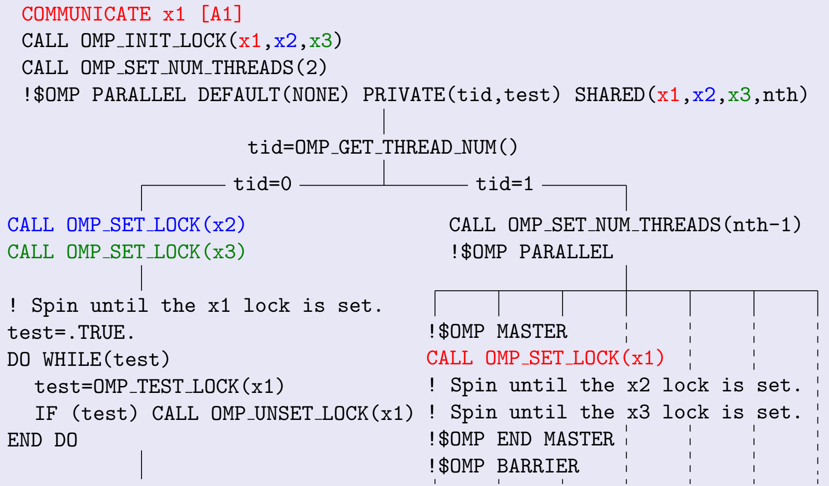
Overcome challenges when using dedicated communication threads:

1. Enforcing the correct sequence of operations in the CCD scheme
2. Shared memory synchronization between the different threads
3. Presumably we need more computation threads than communication threads: how to work-share the computations?

Use OpenMP locks and nested parallelism.

Getting the Routine Started: Setting the OpenMP Locks

- Perform an initial comm. call, and initialize one lock for each direction.
- Always start with two threads: one for comm., one for comput.
- Make sure locks are properly set before performing any operations.



```
COMMUNICATE x1 [A1]
CALL OMP_INIT_LOCK(x1,x2,x3)
CALL OMP_SET_NUM_THREADS(2)
!$OMP PARALLEL DEFAULT(NONE) PRIVATE(tid,test) SHARED(x1,x2,x3,nth)

tid=OMP_GET_THREAD_NUM()
```

```
tid=0

CALL OMP_SET_LOCK(x2)
CALL OMP_SET_LOCK(x3)

! Spin until the x1 lock is set.
test=.TRUE.
DO WHILE(test)
   test=OMP_TEST_LOCK(x1)
   IF (test) CALL OMP_UNSET_LOCK(x1)
END DO
```

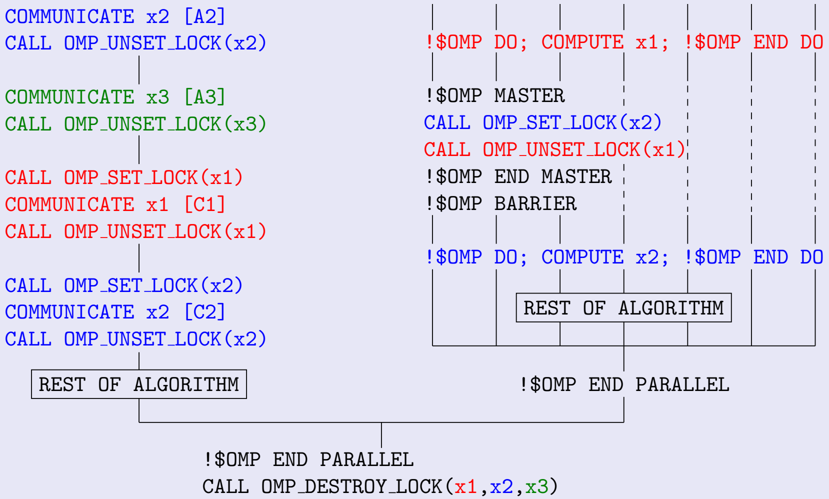
```
tid=1

CALL OMP_SET_NUM_THREADS(nth-1)
!$OMP PARALLEL

!$OMP MASTER
CALL OMP_SET_LOCK(x1)
! Spin until the x2 lock is set.
! Spin until the x3 lock is set.
!$OMP END MASTER
!$OMP BARRIER
```

Working Our Way Through the CCD Scheme

- Obtain lock before operating on a coordinate direction, release when finished (order of comput. thread lock exchange important)



```
COMMUNICATE x2 [A2]
CALL OMP_UNSET_LOCK(x2)

COMMUNICATE x3 [A3]
CALL OMP_UNSET_LOCK(x3)

CALL OMP_SET_LOCK(x1)
COMMUNICATE x1 [C1]
CALL OMP_UNSET_LOCK(x1)

CALL OMP_SET_LOCK(x2)
COMMUNICATE x2 [C2]
CALL OMP_UNSET_LOCK(x2)

REST OF ALGORITHM
```

```
!$OMP DO; COMPUTE x1; !$OMP END DO

!$OMP MASTER
CALL OMP_SET_LOCK(x2)
CALL OMP_UNSET_LOCK(x1)
!$OMP END MASTER
!$OMP BARRIER

!$OMP DO; COMPUTE x2; !$OMP END DO

REST OF ALGORITHM

!$OMP END PARALLEL
```

```
!$OMP END PARALLEL
CALL OMP_DESTROY_LOCK(x1,x2,x3)
```

Scalability of CCD Scheme on (Cray XE6) Blue Waters

- Weak scaling improved from 58% (single-threaded, blocking) to 90% (dedicated comm. thread) for 8192^3 production problem on $256K$ PEs

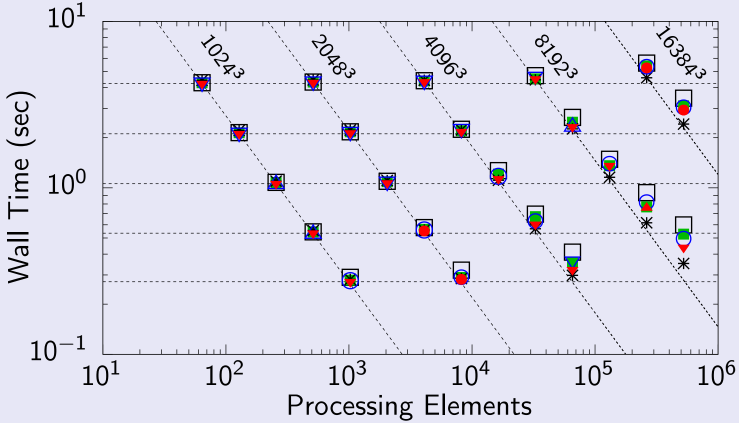


Figure: Timings on BW (Cray XE6) for □ single-threaded, blocking; △▽○ multi-threaded, blocking; ■ single-threaded, overlapped; ▲▼● multi-threaded, overlapped; ∗ one dedicated communication thread per NUMA domain.

Examining Details of Explicit Overlap Approach

- Look at detailed timeline data of comm. and comput. threads
  - Weak-scaling study at target problem density (8192^3 on $8K$ nodes)
  - Record starting and ending times of each operation with `MPI_WTIME()`
  - Red for x_1, blue for x_2, green for x_3, and (thick) black for spinning on a lock

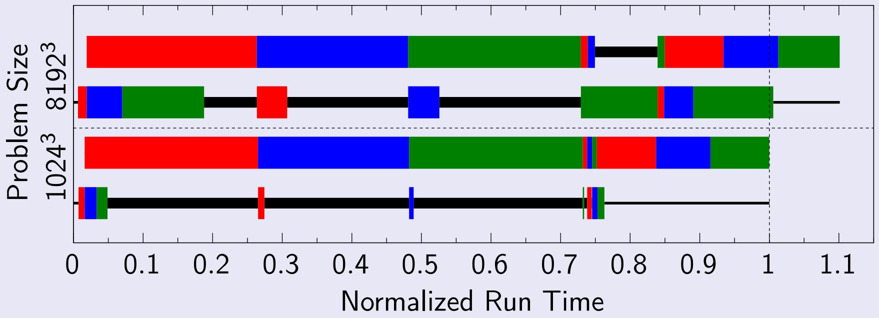


Figure: Thread timeline data. For each problem size, bottom timeline for comm. thread, and top timeline for comput. thread.

A Similar Approach Using Explicit OpenMP Tasks

How else can we express our ideas in the source code?

- Package comm. and comput. operations in explicit OpenMP tasks.
- Enforce order of operations with DEPEND (fully supported in CCE/8.6).
- Two initial threads and nested regions; use MPI_THREAD_SERIALIZED.

```
!$ CALL OMP_SET_NUM_THREADS(2)
!$OMP PARALLEL
!
! Set number of computational threads for all nested regions.
!$ CALL OMP_SET_NUM_THREADS(nth-1)
!
! Master thread generates all tasks (only one shown below).
!$OMP MASTER
! A communication task for the X1 direction.
!$OMP TASK DEPEND(OUT:SCX1,COMM)
CALL MPI_SENDRECV(...)
!$OMP END TASK
!$OMP END MASTER
!$OMP END PARALLEL
```

Outline

Acceleration of Production DNS Code

Challenges associated with porting codes to run on GPUs:

- Maintain maximum code portability by retaining legacy Fortran code base. Rely heavily on OpenMP 4.X offloading available through CCE.
  - Still requires effort to ensure loops achieve high performance on GPUs.
- Severe imbalance between memory available on the host and device.
  - On Titan XK7 nodes, 32 GB available on the host, 6 GB on the GPU.
- Must minimize data movements between the host and device.

Plans and questions for current acceleration effort:

- Overall cost dominated by scalar field computation: accelerate this portion, leave small velocity computation untouched.
- Minimize data movement: put entire scalar computation on the GPU.
- Can scalability be improved by overlapping communication and computation with OpenMP 4.5?

Getting the Process Started

The porting process can be long and complicated, depending on how well-suited the initial algorithm is for acceleration.

- Can the code run on an acceptable number of GPUs efficiently, i.e., with good scaling relative to smaller problem sizes and core counts?
- Are all kernels in the code accelerated well enough? Are there some critical kernels which require special treatment?

Some changes to our code/algorithm were inevitable.

- Important to manage memory: reduce the number of arrays and develop "low-storage" algorithms which still offer good performance.
- MPI on the host uses derived datatypes with strided accesses: must pack data to move between host and device into contiguous buffers.

A Conflict: Memory Layout vs Computational Performance

Code uses 3D arrays which are all allocated in the same way.

- For example: `ALLOCATE(df1(nc1,nc2,nc3))`.
- For most loops, we get great (coalesced) access along the inner index.

A problematic kernel: solving a linear system along the inner index.

```
DO k=1,nc3; DO j=1,nc2; DO i=2,nc1
df1(i,j,k)=F[df1(i,j,k),df1(i-1,j,k)]
END DO; END DO; END DO
```

- Cannot vectorize `i` loop, but need memory access along inner index.

Swap memory layout to improve this kernel:

- Make `j` loop the inner index: `ALLOCATE(buf(nc2,nc1,nc3))`

```
DO k=1,nc3; DO i=2,nc1; DO j=1,nc2
buf(j,i,k)=F[buf(j,i,k),buf(j,i-1,k)]
END DO; END DO; END DO
```

- Not free: loops elsewhere in code need original memory layout.

Improving the Surrounding Loops with Blocking

Need to restrict the changes to memory layout as much as possible

- Use a buffer array and block loops surrounding linear system solution
- For example, the loop that finalizes the solution to CCD now looks like

```
!$OMP TARGET TEAMS DISTRIBUTE COLLAPSE(5)
DO k=1,nc3
   DO bj=0,xbj_max,xbj_siz
      DO bi=0,xbi_max,xbi_siz
         DO j=1,xbj_siz
            DO i=1,xbi_siz
               df1(bi+i,bj+j,k)=F[buf(bj+j,bi+i,k),...]
            END DO
         END DO
      END DO
   END DO
END DO
!$OMP END TARGET TEAMS DISTRIBUTE
```

- Blocking factors tuned to achieve best overall performance.

Performance of Routine Applying CCD in the x_1 Direction

Loop-level performance for CPU and GPU execution

- Focusing on computations in the x_1 direction
- GPU performance metrics with nvprof: `dram_util.`, `alu_fu_util.`
- Test problem: 512^3 with 2x2x2 process layout and 4 OpenMP threads

Computations with original memory layout

Loop	CPU (s)	GPU (s)	Speedup	`dram`	`alu`
RHS	0.0895	0.0125	7.13	7	9
Lin. Sys.	0.5576	0.2161	2.58	2	1
Final Sol.	0.2354	0.0211	11.2	7	8
Total	0.8824	0.2497	3.53	—	—

Computations with swapped memory layout and loop blocking

Loop	CPU (s)	GPU (s)	Speedup	`dram`	`alu`
RHS	0.1265	0.0185	6.84	5	9
Lin. Sys.	0.1407	0.0336	4.19	7	2
Final Sol.	0.1515	0.0153	9.88	8	9
Total	0.4187	0.0674	6.21	—	—

Summary of Acceleration of DNS Code Using CCE/8.6

Algorithmic changes to RK4 required to run on Titan

- For best performance, do not calc. all derivatives together (memory).

Step	Device	Operation Summary
1	CPU	Receive velocity field and fill ghost layers
2	PCI	Transfer u_1 velocity to GPU
3	ALL	Calculate scalar derivatives in x_1; interpolate u_1
4	PCI	Begin transfer of u_3 velocity to GPU
5	GPU	Increment RK4 with x_1 diffusion and partial advection
6	ALL	Calculate advection derivative in x_1
7	GPU	Increment RK4 with x_1 advection term
8	ALL	Calculate scalar derivatives in x_2 and x_3; interpolate u_3
9	PCI	Begin transfer of u_2 velocity to GPU
10	GPU	Increment RK4 with x_2 and x_3 diffusion and x_3 advection
11	ALL	Begin calculation of x_3 advection derivative; interpolate u_2
12	GPU	Increment RK4 with x_2 partial advection
13	ALL	Finalize advection derivatives in x_2 and x_3
14	GPU	Perform RK4 sub-stage update

OpenMP 4.5 Usage with CCE/8.6 in DNS Code

Use tasking clauses on TARGET construct to overlap comm./comput.

- Ensure correct ordering of kernels with DEPEND and a directionally-dependent dummy variable, e.g., SYNCX3 for the x_3 direction.
- Before performing communication in, say, x_3, launch all available x_2 kernels asynchronously with NOWAIT.

```
! From previous data movement, make sure data is on host.
!$OMP TARGET DEPEND(IN:SYNCX3)
!$OMP END TARGET
!
! Launch all kernels in the X2 direction (showing just one).
!$OMP TARGET TEAMS DISTRIBUTE DEPEND(INOUT:SYNCX2) NOWAIT
<Computational task on the GPU for the X2 direction>
!$OMP END TARGET TEAMS DISTRIBUTE
!
! Proceed with communication call in the X3 direction.
CALL MPI_ALLTOALL(...)
```

Performance and Scalability of Accelerated DNS Code

Appx. 5X speedup, with improvement from 75% (non-async.) to 89% (async. with `NOWAIT`) weak-scaling for 8192^3 on $8K$ nodes.

CPU-only:	15.7 for 512^3 and 16.5 for 8192^3
OpenMP 4.5 GPU:	2.93 for 512^3 and 3.30 for 8192^3

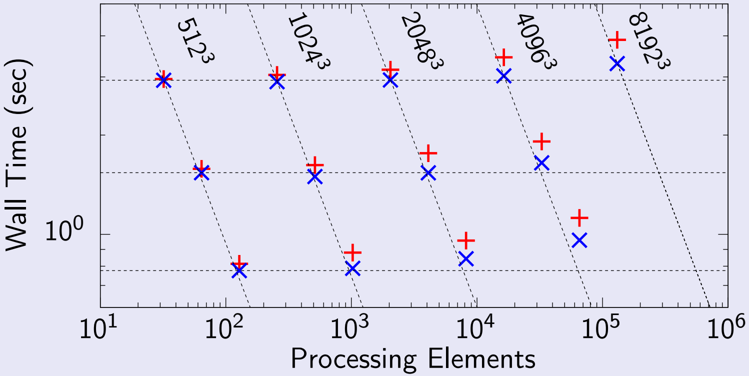


Figure: GPU code timings for non-async. (+) and async. using `NOWAIT` (X).

Outline

OpenMP Wish List and Feedback

OpenMP has been extremely useful to achieve high performance and good scalability on homogeneous and heterogeneous architectures.

1. We could use non-contiguous movements between host and device:

```
!$OMP TARGET UPDATE FROM(us(1:2,:,:))
```

2. More intuitive synchronization constructs for a given “stream”:

```
!$OMP TASKWAIT DEPEND(SYNC) <-> !$OMP TARGET DEPEND(IN:SYNC)
                                !$OMP END TARGET
```

3. We would like to time asynchronous TARGET tasks without using system software, perhaps with additional OpenMP clauses:

```
!$OMP TARGET TEAMS DISTRIBUTE DEPEND(OUT:SYNC) NOWAIT &
!$OMP START_TIME(T1) FINISH_TIME(T2)
<Computational kernel>
!$OMP END TARGET TEAMS DISTRIBUTE
```

Conclusions and Future Work

Using OpenMP in a petascale turbulence code:

- On CPUs (BW): dedicated comm. threads improve scalability
- On GPUs (Titan): OpenMP 4.5 with CCE/8.6 giving high performance
  - Algorithmic changes to reduce memory
  - Overlap comm. and comput. with `NOWAIT` and `DEPEND`

Future work:

- Manuscript being prepared to report algorithms and performance
- Acceleration of pseudo-spectral scheme may be of more general interest
- Working on port of kernels to prototype architecture for Summit
  - Transitioning from Cray's CCE to IBM's XLF: performance portability
  - Different node architecture: multiple GPUs on each node