Challenges in particle tracking in turbulence on a massive scale

Dhawal Buaria ¹  P. K. Yeung ¹

¹Georgia Institute of Technology

Supported by NSF (CBET-1235906)
Supercomputing resources: NCSA (PRAC), TACC (XSEDE)

XSEDE 2014
Atlanta, GA, July 13-18, 2014
Introduction

Turbulence and Direct Numerical Simulations (DNS)

Particle tracking and 3-D interpolation

Parallel implementation

Hybrid MPI/OpenMP implementation

Using PGAS programming model (CoArray Fortran)

Conclusions and Future Work
Introduction: What is Turbulence?

- Most common state of fluid motion in nature and engineering
- Characterized by disorderly fluctuations in time and 3-D space
- Wide range of non-linearly interacting scales, usually at high Reynolds number \((Re = UL/\nu)\)
Turbulence and DNS

- Wide range of scales depending strongly on Reynolds number
  - domain size \( (L_0) \) larger than largest scale \( L \)
  - grid spacing \( (\Delta x) \) comparable to smallest scale \( (\eta) \)
  - classical scaling \( L/\eta \sim R_L^{3/4} \), where \( R_L = uL/\nu \)

- Range of time scales: \( (L/u)/\tau_\eta \sim R_L^{1/2} \)

- Direct Numerical Simulations (DNS):
  - resolve all spatial and temporal scales
  - many grid points \( (N^3 \sim R_L^{9/4}) \) and many time steps
  - tremendous details (beyond experiments)
  - computationally very expensive (total cost \( \sim R_L^3 \))

- Need efficient numerical methods and parallel algorithms

- Data storage and postprocessing also a major challenge
Eulerian vs. Lagrangian frame of reference

**Eulerian:**
- Fixed observer in lab frame
- Treat fluid motion as whole and depict flow quantities as a function of position $x$ and time $t$
- Solve the governing equations for a fixed set of grid points ($N^3$) and large number of time steps

**Lagrangian:**
- Observer follows an individual fluid ‘particle’ as it moves through space and time
- Fluid ‘particle’ has zero size and moves with local flow velocity
- Important in studying applications like pollutant dispersion, cloud physics, mixing, etc.
- Similar approach to study molecular motion, interial particles, etc.
Governing equations and Numerical approach

- DNS of isotropic turbulence in a periodic cubic domain
- The Eulerian flow velocity \( \mathbf{u}(\mathbf{x}, t) \) is obtained by solving the Navier-Stokes equations with constant density \( \nabla \cdot \mathbf{u} = 0 \)

\[
\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla \left( \frac{p}{\rho} \right) + \nu \nabla^2 \mathbf{u} + \mathbf{f}
\]

- Equations are solved in Fourier space using a pseudo-spectral scheme (3D-FFT as code kernel)
- Advance in time by 2nd or 4th order explicit Runge-Kutta; \( \Delta t \) based on CFL number (for numerical stability)
- 2-D domain decomposition, 3 X 1D-FFT, 2 X ALLTOALL : (Donzis et al., Turbulence simulations on \( O(10^4) \) processors, Proc. TeraGrid 2008 Conference)
Particle tracking and interpolation

- Lagrangian approach to study turbulent dispersion
- Large number of fluid particles introduced at $t = 0$ in the domain
- Integrate $dx^+ / dt = u^+$, where $+$ denotes Lagrangian quantity
- $u^+$ is the velocity at particle position $x^+$: $u^+ = u(x^+, t)$
- Use cubic-spline interpolation: 4th order accurate, twice differentiable; useful for computing velocity gradients and Laplacian also

$$
u^+ = \sum_k \sum_j \sum_i b_i(x^+) c_j(y^+) d_k(z^+) e_{ijk}(x)$$

where $(b_i, c_j, d_k)$ are basis functions at 4 adjacent grid intervals, and $(e_{ijk})$ are $(N + 3)^3$ Eulerian spline coefficients (Yeung and Pope JCP 1988, Yeung and Pope JFM 1989)
Parallel Implementation for Particle Tracking

- Large number of fluid particles (up to tens of millions)
- Divided equally among all MPI tasks
- Particles wander randomly
- Neighboring grid points for interpolation typically on different task
Steps for Interpolation

Recall: $u^+ = \sum_k \sum_j \sum_i b_i(x^+)c_j(y^+)d_k(z^+)e_{ijk}(x)$

Interpolation done in three steps

1. Compute the 3-D spline coefficients $(e_{ijk})$ from the Eulerian velocity; involves 3 computation cycles and 2 transposes (ALLTOALLVs); independent of number of particles, can be reused

2. Compute the basis functions $(b_i, c_j, d_k)$ locally on each task; use MPI_ALLGATHER to collect them on all tasks (requires lot of memory, so process in smaller batches)

3. Scan through all particles and collect partial sums on each task; MPI_REDUCE (using MPI_SUM) to collect all sums and then MPI_SCATTER to send them to respective tasks.

All steps communication dominant
Hybrid MPI/OpenMP approach

- Due to random nature of particle movement, neighboring interpolation stencil almost always on different MPI tasks
- More than 90% time in particle tracking spent on MPI communication at 16k cores (even worse for more cores)
- Hybrid MPI/OpenMP paradigms reduce latency effects
- Number of MPI tasks reduced by number of OpenMP threads; also provides more memory per MPI task
- Use MPI\_THREAD\_FUNNELED; master thread communicates
- Usually benefits the computational part also
Performance results on BlueWaters, NCSA

Timings on BlueWaters (BW), NCSA using 512 nodes (16384 cores)

$4096^3$ grid points, 16 million particles

<table>
<thead>
<tr>
<th>Proc. grid</th>
<th>32x512</th>
<th>16x512</th>
<th>8x512</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of threads</td>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>MPI_Allgather</td>
<td>0.847</td>
<td>0.704</td>
<td>0.428</td>
</tr>
<tr>
<td>Computations</td>
<td>0.236</td>
<td>0.117</td>
<td>0.071</td>
</tr>
<tr>
<td>Reduce+Scatter</td>
<td>1.588</td>
<td>1.067</td>
<td>0.907</td>
</tr>
<tr>
<td>Total</td>
<td>2.671</td>
<td>1.888</td>
<td>1.406</td>
</tr>
</tbody>
</table>

- Better performance with more threads (can’t go above 4 threads ?)
- Performance also depends on network topology and contention; best performance when running in isolated cabinet(s) with no interference from other jobs (reservation on request)
Performance results on BW (increasing particles)

- Recall processor grid \( P = P_1 \times P_2 \) set by Eulerian part
- Best performance for: \( P_1 \times \text{no. of threads} = \text{cores on single node} \) (32 for BlueWaters); also required that \( P_1 \geq \text{no. of threads} \)
- How does the code scale with more particles?

Timings on BW, 4096\(^3\) grid points, 512 nodes (16384 cores), 8x512, 4 threads

<table>
<thead>
<tr>
<th>No. of particles</th>
<th>16M</th>
<th>32M</th>
<th>64M</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allgather</td>
<td>0.505</td>
<td>0.755</td>
<td>1.476</td>
</tr>
<tr>
<td>Computation</td>
<td>0.068</td>
<td>0.137</td>
<td>0.272</td>
</tr>
<tr>
<td>Reduce+Scatter</td>
<td>1.012</td>
<td>1.759</td>
<td>3.478</td>
</tr>
<tr>
<td>Total</td>
<td>1.585</td>
<td>2.651</td>
<td>5.226</td>
</tr>
</tbody>
</table>

- Similar improvement on other machines.
Performance results on BW (increasing particles)

- Recall processor grid \((P = P_1 \times P_2)\) set by Eulerian part
- Best performance for: \(P_1 \times \) no. of threads = cores on single node (32 for BlueWaters); also required that \(P_1 \geq\) no. of threads
- How does the code scale with more particles?

Timings on BW, \(4096^3\) grid points, 512 nodes (16384 cores), \(8\times512, 4\) threads

<table>
<thead>
<tr>
<th>No. of particles</th>
<th>16M</th>
<th>32M</th>
<th>64M</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allgather</td>
<td>0.505</td>
<td>0.755</td>
<td>1.476</td>
</tr>
<tr>
<td>Computation</td>
<td>0.068</td>
<td>0.137</td>
<td>0.272</td>
</tr>
<tr>
<td>Reduce+Scatter</td>
<td>1.012</td>
<td>1.759</td>
<td>3.478</td>
</tr>
<tr>
<td>Total</td>
<td>1.585</td>
<td>2.651</td>
<td>5.226</td>
</tr>
</tbody>
</table>

- Similar improvement on other machines.
Performance results on Stampede (TACC)

$2048^3$ grid points, 16M particles, 128 nodes (2048 cores)

<table>
<thead>
<tr>
<th>Proc. Grid</th>
<th>16x128</th>
<th>4x128</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of threads</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>MPI_Allgather</td>
<td>0.458</td>
<td>0.373</td>
</tr>
<tr>
<td>Computation</td>
<td>0.282</td>
<td>0.182</td>
</tr>
<tr>
<td>Reduce+Scatter</td>
<td>1.249</td>
<td>0.944</td>
</tr>
<tr>
<td>Total</td>
<td>1.989</td>
<td>1.499</td>
</tr>
</tbody>
</table>

Going to larger problem size and more particles

$4096^3$ grid points, using 1024 nodes (16384 cores), 8x1024, 2 threads

<table>
<thead>
<tr>
<th>No. of particles</th>
<th>16M</th>
<th>64M</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allgather</td>
<td>0.616</td>
<td>2.262</td>
</tr>
<tr>
<td>Computations</td>
<td>0.162</td>
<td>0.502</td>
</tr>
<tr>
<td>Reduce+Scatter</td>
<td>1.005</td>
<td>4.662</td>
</tr>
<tr>
<td>Total</td>
<td>1.783</td>
<td>7.426</td>
</tr>
</tbody>
</table>
## Performance results on Stampede (TACC)

2048\(^3\) grid points, 16M particles, 128 nodes (2048 cores)

<table>
<thead>
<tr>
<th>Proc. Grid</th>
<th>16x128</th>
<th>4x128</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of threads</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>MPI_Allgater</td>
<td>0.458</td>
<td>0.373</td>
</tr>
<tr>
<td>Computation</td>
<td>0.282</td>
<td>0.182</td>
</tr>
<tr>
<td>Reduce+Scatter</td>
<td>1.249</td>
<td>0.944</td>
</tr>
<tr>
<td>Total</td>
<td>1.989</td>
<td>1.499</td>
</tr>
</tbody>
</table>

- Going to larger problem size and more particles

4096\(^3\) grid points, using 1024 nodes (16384 cores), 8x1024, 2 threads

<table>
<thead>
<tr>
<th>No. of particles</th>
<th>16M</th>
<th>64M</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allgater</td>
<td>0.616</td>
<td>2.262</td>
</tr>
<tr>
<td>Computations</td>
<td>0.162</td>
<td>0.502</td>
</tr>
<tr>
<td>Reduce+Scatter</td>
<td>1.005</td>
<td>4.662</td>
</tr>
<tr>
<td>Total</td>
<td>1.783</td>
<td>7.426</td>
</tr>
</tbody>
</table>
PGAS = Partitioned Global Address Space

- Instead of MPI, can use PGAS programming model to do communication.
- FORTRAN + CoArrays: allocated in ‘global memory address space’, can be accessed by all ‘images’ (MPI tasks).
- Currently fully supported only on Cray compiler and Gemini network.
- CAF has one-sided communication, lower latency, smaller headers.
- Replaced MPI_ALLTOALL(V) calls by library routines in CAF; generated with help from consultants at NCSA/CRAY.
- Library routine copies messages to/from statically allocated co-array 'buffer' on each image; breaks messages into small chunks.
- Pulls chunks from other images in random order (reduces network congestion).
- Reduces overall time by ∼ 33% on 4096 nodes (BW, NCSA).

Consider the REDUCE + SCATTER model in last step of interpolation
  - too much traffic to and from one particular MPI task
  - not the most efficient pathing for data movement

Instead can use a binary tree type algorithm, best implemented by using Co-Array Fortran (CAF)

Images (MPI tasks) form mutually exclusive pairs to exchange information both relevant to themselves and images they subsequently communicate with

$log_2 P$ steps required, where $P =$ total number of images; diminishing message size for each step
Consider a case with $P = 8$; $\log_2 P = 3$ required steps.

Arrow indicates direction of movement followed by summation.

Task pairs: $(0,1)$ $(2,3)$ $(4,5)$ $(6,7)$

Task pairs: $(0,2)$ $(1,3)$ $(4,6)$ $(5,7)$
Schematic of the algorithm (contd.)

Step 3

- task 0
- task 4
- task 1
- task 5
- task 2
- task 3
- task 7
- task 6

Final Layout

- task 0
- task 1
- task 2
- task 3
- task 4
- task 6
- task 7
- task 5

Pairs: (0,4) (1,5) (2,6) (3,7)

- Communication pattern always evenly distributed; no congestion to any particular image
### Performance on BW, $4096^3$, 16M particles, 16k cores

<table>
<thead>
<tr>
<th></th>
<th>MPI</th>
<th>MPI</th>
<th>CAF</th>
<th>CAF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc. Grid</td>
<td>32x512</td>
<td>8x512</td>
<td>32x512</td>
<td>8x512</td>
</tr>
<tr>
<td>No. of threads</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>MPI_Allgather</td>
<td>1.810</td>
<td>0.861</td>
<td>1.789</td>
<td>0.833</td>
</tr>
<tr>
<td>Computation</td>
<td>0.498</td>
<td>0.144</td>
<td>0.488</td>
<td>0.144</td>
</tr>
<tr>
<td>Reduce+Scatter</td>
<td>3.648</td>
<td>1.956</td>
<td>2.252</td>
<td>0.905</td>
</tr>
<tr>
<td>Total</td>
<td>5.956</td>
<td>2.961</td>
<td>4.529</td>
<td>1.882</td>
</tr>
</tbody>
</table>

- **4X** speedup for REDUCE+SCATTER using OpenMP and CAF
Conclusions and Future Work

- Hybrid MPI/OpenMP implementation for parallel interpolation helps to improve both communication and computation (though code is communication dominant)
- Overall time reduced by almost 50% on BW when using 4 threads
- Scales well with increased number of particles
- Further improvement in communication timings by using CAF

Future Work:

- Extend scaling to even larger problems. Working on $8192^3/16M$ particles using 8192 nodes (262144 cores) on BW
- Working on alternate algorithm to do the interpolation, where each set of particles go through each task one by one in a pipelined fashion
- Extend to utilize MIC architecture on Stampede (recently applied for a ECSS project)