Spectral Navier Stokes Solver with UPC++ and OpenMP

Samuel Olivier
`https://github.com/smsolivier/spectral_upcxx`

May 6, 2018

1 Background

The Navier Stokes Equations are:

$$\frac{\partial \vec{V}}{\partial t} = \vec{V} \times \vec{\omega} - \nabla \Pi + \nu \nabla^2 \vec{V}\,, \tag{1.1a}$$

$$\nabla \cdot \vec{V} = 0\,, \tag{1.1b}$$

where $\vec{V}$ is the velocity, $\vec{\omega} = \nabla \times \vec{V}$ the vorticity, $\Pi = \frac{1}{2}\vec{V}^2 + \frac{P}{\rho}$ a pressure-like term, and ν the viscosity. Equations 1.1 will be solved *spectrally* in a triply-periodic box $[0, 2\pi]^3$ by transforming all variables into Fourier space:

$$f(x, y, z) = \sum_m \sum_n \sum_p \tilde{f}(m, n, p) e^{imx} e^{iny} e^{ipz}\,, \tag{1.2}$$

where $\tilde{f}(m, n, p)$ are the Fourier coefficients. In Fourier space, many operators are now trivial. For example, the gradient is:

$$\nabla f(x, y, z) = \sum_m \sum_n \sum_p (im\hat{x} + in\hat{y} + ip\hat{z}) \tilde{f}(m, n, p) e^{imx} e^{iny} e^{ipz}\,. \tag{1.3}$$

In addition, the orthogonality of the Fourier modes decouples the Fourier modes in m, n, p such that each Fourier mode can be independently evolved in time. A fractional step method employing second order Adams Bashforth (AB2) and Crank Nicholson (CN) will be used:

$$\tilde{V}^{n+1/2} = \tilde{V}^n + \frac{\Delta t}{2}\left(3\tilde{V}^n \times \tilde{\omega}^n - \tilde{V}^{n-1} \times \tilde{\omega}^{n-1}\right) - \nabla \tilde{G}\,, \tag{1.4a}$$

$$\tilde{V}^{n+1} = \tilde{V}^{n+1/2} + \frac{\nu \Delta t}{2}\left(\nabla^2 \tilde{V}^{n+1} + \nabla^2 \tilde{V}^n\right)\,, \tag{1.4b}$$

where $\tilde{G} = \frac{\Delta t}{2}\left(3\Pi^n - \Pi^{n-1}\right)$ is found by enforcing $\nabla \cdot \tilde{V}^{n+1} = 0$:

$$\nabla^2 \tilde{G} = \nabla \cdot \left[\tilde{V}^n + \frac{\Delta t}{2}\left(3\tilde{V}^n \times \tilde{\omega}^n - \tilde{V}^{n-1} \times \tilde{\omega}^{n-1}\right) + \frac{\nu \Delta t}{2} \nabla^2 \tilde{V}^n\right]\,. \tag{1.5}$$

Note that $\tilde{V} \times \tilde{\omega}$ requires an expensive Fourier convolution. To avoid this, the cross product is computed in physical space. In other words, inverse transform $\tilde{V}$ and $\tilde{\omega}$ to physical space, compute the cross product, and transform the result into Fourier space. Thus, this algorithm is highly dependent on the Fast Fourier Transform (FFT).

2 Implementation

The implementation is centered around the distributed array class, `Scalar`. `Scalar` stores a 3D array distributed across UPC++ ranks in the z dimension. This was accomplished by creating a vector of `upcxx::global_ptr` to `std::complex<double>` and broadcasting each rank's pointer to all other ranks.

The 3D FFT is built around this data structure. One-dimensional, serial FFTW [1] is used to transform each dimension. A custom wrapper class, `FFT1D`, handles FFTW plan creation, destruction, and execution.

Since the array is only distributed in z, x and y are locally owned. All the local columns are transformed independently and then the x rows are transformed. A global transpose is used to switch the data to be distributed in y such that x and z are now local. Serial FFTW can then be used in the z direction. The array is then transformed back to be distributed in z. The global transpose is performed out of place and thus requires twice as much memory.

In addition, operators such as gradient, Laplacian, inverse Laplacian, and element-wise arithmetic between `Scalar`'s are defined. These operations are all element-wise and do not require communication.

`Scalar` also stores flags for whether the data is in physical or Fourier space. All functions have checks to ensure operations begin in the correct space to avoid user errors in the main function. This also helps in outputting to VTK as all variables are automatically transformed out of place to physical space, if required.

Vector fields are represented with the `Vector` class which simply stores three `Scalars` in an array. As with `Scalar`, common operators are defined such as curl, divergence, vector Laplacian, arithmetic, and cross product. Transforming a `Vector` simply calls the `Scalar` transform on each component.

Through abstraction and operator overloading of arithmetic and indexing Eqs. 1.4a and 1.4b are simply:

```
Vhalf = V1 + dt/2*(3*V1.cross(omega1) - V0.cross(omega0)) - G.gradient();
V = Vhalf + nu*dt/2*V1.laplacian();
for (int d=0; d<DIM; d++) {
   V[d].laplacian_inverse(1, -nu*dt/2); // invert each component
}
```

Thus, all the parallel aspects are hidden from the main program.

The `Writer` class handles parallel output to VTK. `Writer` stores pointers to `Vector`'s and `Scalar`'s and outputs their data after a given number of time steps. Using `Scalar`'s `isFourier` flag, inverse transforms are called automatically if a variable is in Fourier space.

`Writer` uses `VisitWriter` to write each rank's local data to VTK. A Visit master file is created to combine the decomposed files in Visit [2]. Unfortunately, it was not possible to get Visit to connect the decomposed files so they appear as if they were one file. Instead, the output has gaps in the z direction.

`CH_Timer` (Chombo timer) was used to manually instrument the code.

3 Optimizations

OpenMP In an attempt to reduce communication, OpenMP threads were used for all local computations. Since OpenMP uses shared memory, fewer messages would need to be sent in the global transpose while still maintaining the same level of parallelism. The outermost loop of all the 1D transforms, operators, and arithmetic were done in parallel with `#pragma omp parallel for`.

Pencils and Slabs Computation and communication were interleaved by sending messages during the second phase of 1D transforms. Two message sizes were implemented: pencils and slabs. Pencils sends each row immediately after it is transformed while slabs transforms a chunk of rows and then sends them all off at once. Both methods use one-sided communication with `upcxx::rput`.

Combining OpenMP and pencils required sending messages from an OpenMP parallel region. While implementing this, I discovered a bug in the Cori installation of UPC++ and reported it through the UPC++ Google Group. Adding `upcxx::default_persona_scope()` at the top of every OpenMP parallel region was used as a temporary workaround.

FFTW_MEASURE At first, FFTW plans were re-made for each transform. By altering `FFT1D` to store FFTW plans, plans could be made at initialization and reused on different arrays with `fftw_execute_dft` which takes a plan and a pointer that does not need to match the pointer given in the creation of the FFTW plan. Thus, one plan could be re-used on different data. This allowed for a switch to `FFTW_MEASURE` instead of `FFTW_ESTIMATE` which experiments with different algorithms (thus taking longer than `FFTW_ESTIMATE`) to increase performance.

Transpose to z Contiguous In the third phase of transforms (over z), the 1D FFT has a stride of $N_x \times \frac{N_y}{n_{\text{upcxx}}}$. This leads to very poor cache performance. An out-of-place transpose was implemented to switch the data from contiguous in x to contiguous in z. This improved performance of the z FFTs but the transpose itself was too expensive to be viable.

Memcpy At any time when `upcxx::rput` was called to send a message to the same rank, `memcpy` was used instead. This did not significantly alter run times.

Pass by Reference Passing all `Scalar`'s and `Vector`'s by reference to functions prevented calls to the copy constructor. This is important because allocating memory with `upcxx::new_array` is often slow especially for very large arrays. The implementation was helped by creating `const` member functions and defining the copy constructor and copy assignment for `Scalar`.

Preallocate Transpose Memory In scaling studies, the allocation of the temporary memory for the out-of-place global transpose at the beginning of every transform hindered OpenMP scaling as it is a serial operation. Preallocating this temporary memory in the initialization of a `Scalar` removed this bottleneck.

However, this means that every `Scalar` uses twice as much memory throughout its lifetime instead of only during the transform. Since multiple transforms can not be performed in parallel, allocating and deallocating the temporary memory at the beginning and end of every transform means that at any given time only one `Scalar` object could be allocating twice its normal memory. Thus, if memory is limited, this optimization cannot be used. Preallocation also doubles the time to initialize a `Scalar` as both the primary and temporary arrays must be allocated.

Save Intermediate Computations By saving expensive intermediate computations, the number of calls to the cross product and vector laplacian were significantly reduced.

Maximize OpenMP Loop Size For OpenMP scaling, indexing contiguously was ignored if more parallelism could be exposed by switching the order of the loops. For example, in computing the FFT's in the z direction, the loop order was reversed so that the outer most loop was over x instead of y.

4 Performance

4.1 FFT Heterogeneous Scaling

Figure 4.1 shows timings of the three phases of the transform (columns, rows, z), the transpose back, and the total wall time for an FFT of size 512^3. The number of UPC++ ranks was varied across runs and the number of OpenMP threads was chosen such that $n_{\text{upcxx}} \times n_{\text{omp}} = 64$. Pencils and slabs are compared. All data points were run on 2 nodes of Cori Haswell. Since the number of cores is not altered, all runs should have similar timings if scaling issues were absent.

Except for $n_{\text{omp}} = 32$ where NUMA effects may be present, OpenMP performs the same or slightly better in rows and transpose back but fluctuates for columns.

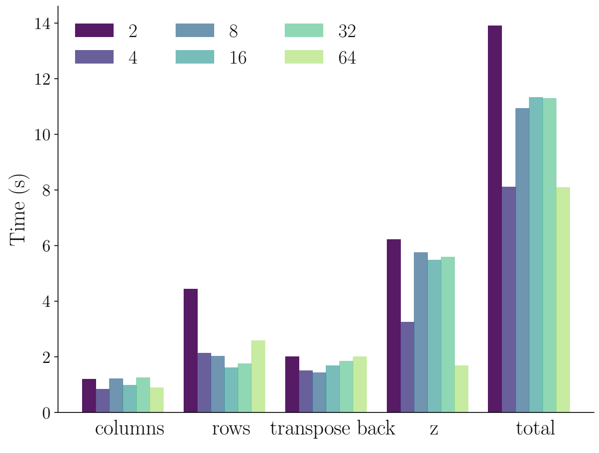
The transforms over z are much slower due to the longer stride of $N_x \times \frac{N_y}{n_{\text{upcxx}}}$. The z transforms had the most variability and no combination of OpenMP performed better than UPC++ alone. This could be from cache effects as increasing n_{upcxx} decreases the stride. Data contention may also be an issue as all the FFT's are performed on the same array.

For both pencils and slabs, OpenMP became viable when $n_{\text{omp}} = 16$, $n_{\text{upcxx}} = 4$. Note that this was only possible with both preallocating the temporary transpose memory and switching the loop ordering of the z FFTs.

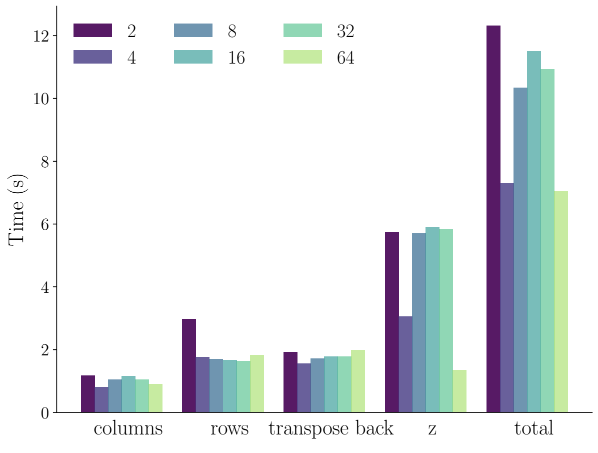
However, this result is problem dependent. Figure 4.2 repeats this experiment on an FFT of size 256^3 which shows more variability in pencils v slabs and a different optimal balance between OpenMP and UPC++.

4.2 Navier Stokes Strong Scaling

The Navier Stokes program was run on a mesh of size 64^3 for a range of UPC++ ranks/OpenMP threads on a single node of Cori Haswell. Figure 4.3a shows the speedup of the most expensive kernels over a range of UPC++ ranks. This is repeated for OpenMP threads in Fig. 4.3b. Note that these plots are relative to the parallel code with 1 rank/thread and not a serial version.

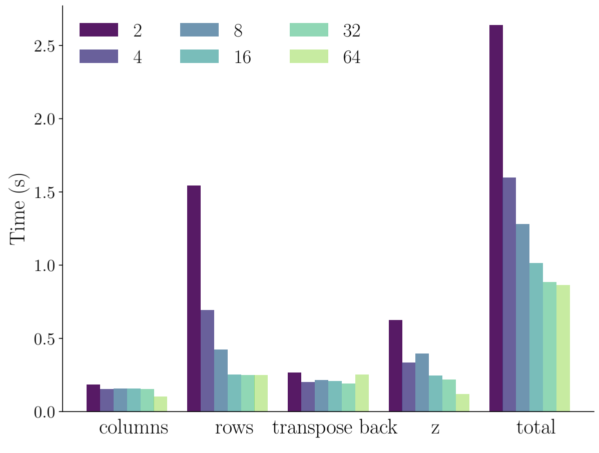


(a)

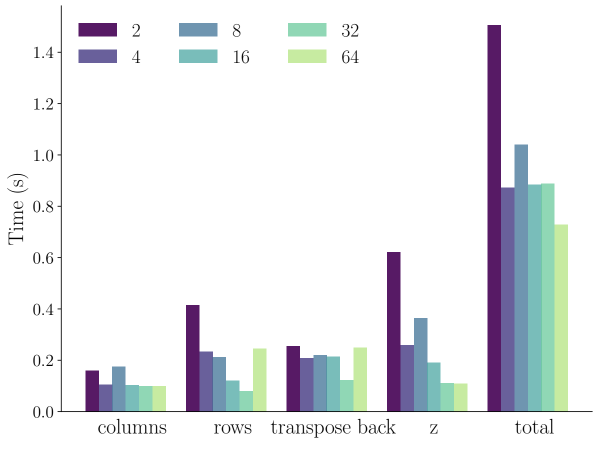


(b)

Fig. 4.1. Wall times for (a) pencils and (b) slabs over a range of UPC++ ranks where $n_{\mathrm{omp}} \times n_{\mathrm{upcxx}} = 64$. FFT size: 512^3.

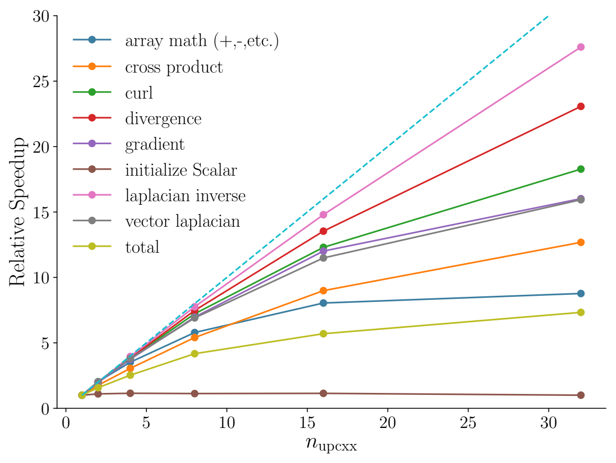


(a)

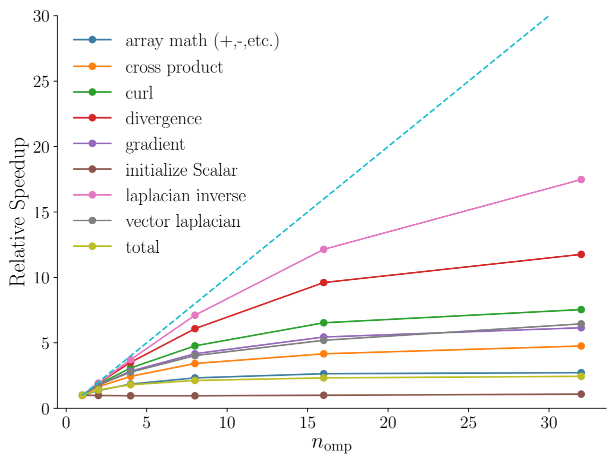


(b)

Fig. 4.2. Wall times for (a) pencils and (b) slabs over a range of UPC++ ranks where $n_{\mathrm{omp}} \times n_{\mathrm{upcxx}} = 64$. FFT size: 256^3.



(a)



(b)

Fig. 4.3. Speedup (relative to the parallel code with one rank/thread) for a range of (a) UPC++ ranks and (b) OpenMP threads.

In both cases, the inverse Laplacian operator strong scaled the best. Of the operators listed, only the inverse Laplacian has a `void` return type. The rest rely on returning a `Vector` or `Scalar` and thus need to allocate memory by calling `Scalar::initialize` which did not scale.

This affects scaling especially when the amount of computation is small. For example, the array math operators spent $\sim 25\%$ of the their time initializing memory for one thread and $\sim 80\%$ initializing for 32 threads. Due to this, array math scaled poorly.

On the other hand, the divergence operator spends three times less initializing memory since it returns a `Scalar` instead of a `Vector`. It's shape more closely follows the inverse Laplacian suggesting it is not as plagued by the initialization issue.

The total curves appear to be limited by the cross product and array math for UPC++. For OpenMP, array math was clearly the bottleneck.

Figure 4.4 shows the speedup of the Navier Stokes compared to a serial version of the code. With 32 ranks, UPC++ only had a speedup of 5 while OpenMP was only 1.5.

The experiment in §4.1 was repeated on a single node of Cori Haswell ($n_{\text{upcxx}} \times n_{\text{omp}} = 32$) for the Navier Stokes code on a mesh of size 64^3. Figure 4.5a shows the timings over a range of n_{upcxx} which shows the same trends as the isolated scaling. Figure 4.5b shows that no combination of OpenMP and UPC++ was able to perform better than UPC++ alone.

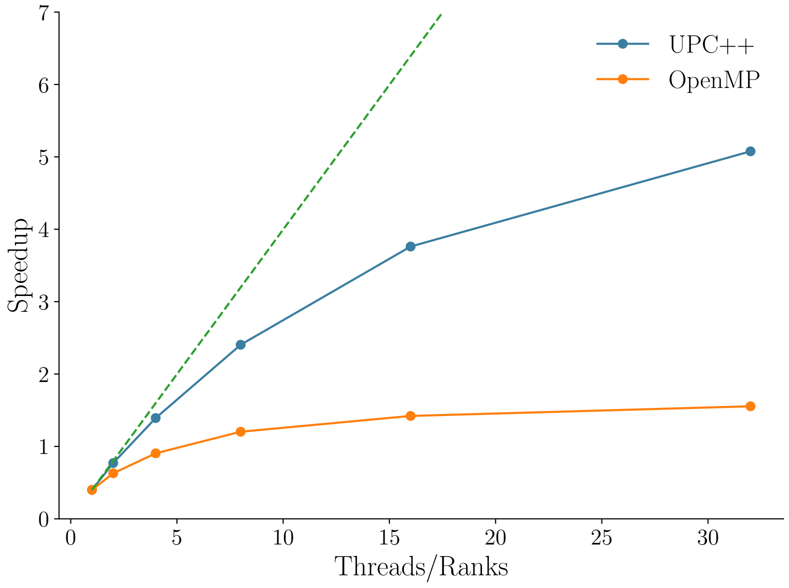
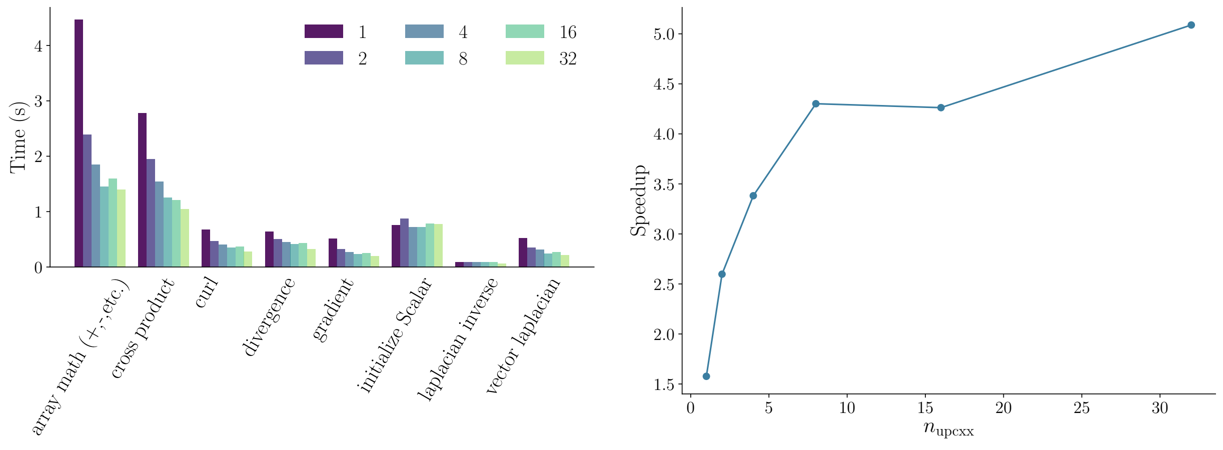


Fig. 4.4. The parallel speedup for UPC++ and OpenMP as compared to the serial code. Mesh size 64^3.



(a)

(b)

Fig. 4.5. Timing breakdown (a) and total speedup (b) for heterogeneous scaling on one node.

5 Application

The code was used to simulate two vortices in a triply-periodic box. The mesh size was 64^3. The simulation was evolved for 12 s with a time step of 1×10^{-3} s. The Reynold's number was $\sim 6 \times 10^4$. Figure 5.1 shows the evolution of the velocity, vorticity, and pressure. One node of Cori Haswell was used.

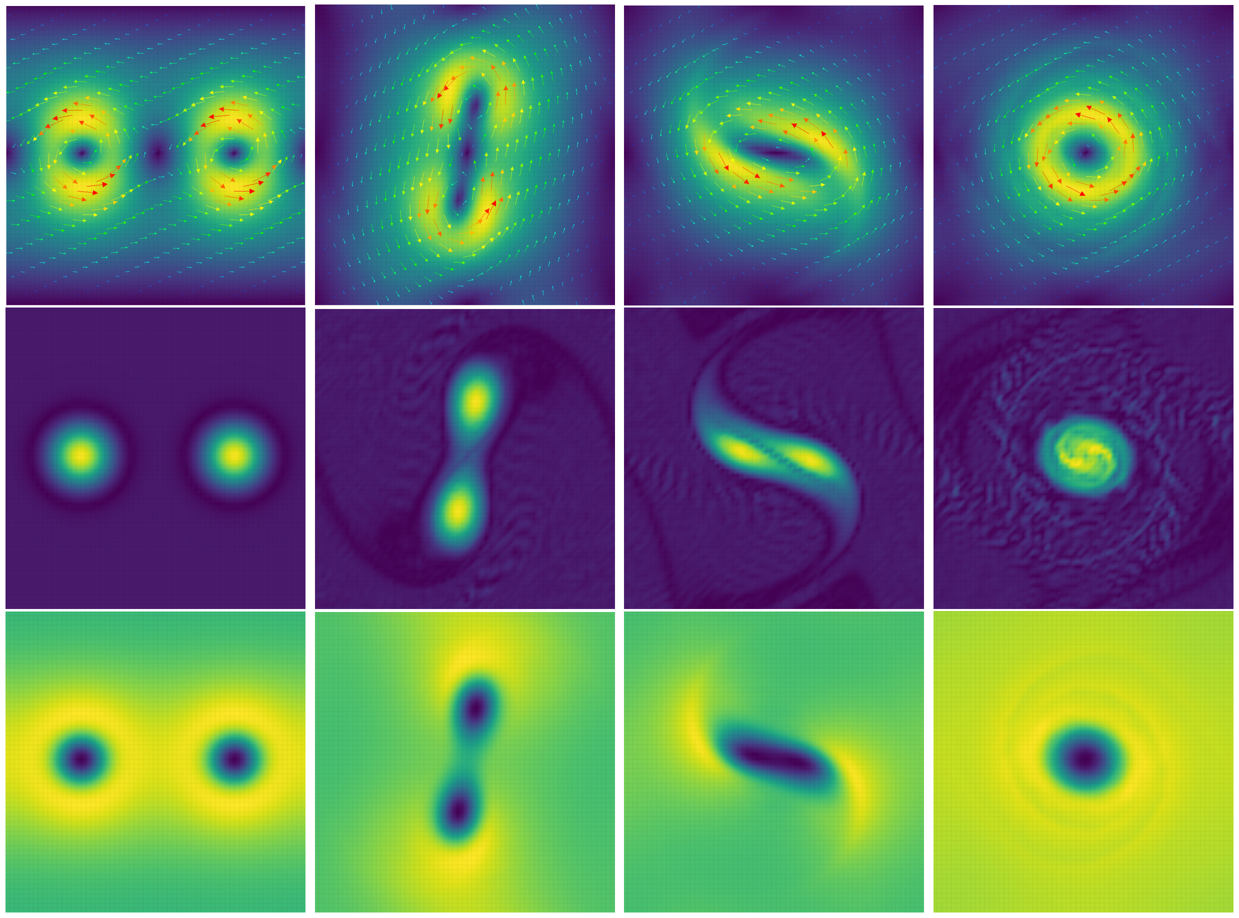


Fig. 5.1. Four time slices of the velocity (top), vorticity (middle), and pressure (bottom).

6 Conclusions

A spectral Navier Stokes solver was implemented with UPC++ and OpenMP. A parallel 3D Fast Fourier Transform was built around a custom, distributed UPC++ data structure that also enabled parallel computation of all the operators needed for the Navier Stokes simulation.

The code successfully employed abstraction and operator overloading to provide a simple API for combining operators into the numerical scheme. In addition, much of the parallelism was hidden behind the scenes.

Combining OpenMP and UPC++ proved to not be as performant as initially believed. Preallocating the global transpose memory and properly ordering the OpenMP parallel loops led to the possibility of OpenMP/UPC++ performing as well as UPC++ alone (after careful tuning of the balance between the two). It is possible that even larger problems requiring more inter-node communication may benefit from inter-opting with OpenMP as well as Cori KNL.

However, this was not possible in other operations such as array arithmetic and cross product due to the poor scaling of initializing memory with `upcxx::new_array`. This especially affected OpenMP scaling since the size of the allocation does not change with increasing OpenMP threads. UPC++ alone performed better than any combination of OpenMP and UPC++ in the Navier Stokes simulation.

The frequent need for memory initialization arose from the use of return types motivated by increasing abstraction and ease of use. This was especially true for operator overloaded arithmetic such as +, -, and * that took in `&Scalar/&Vector` and returned a newly allocated `Scalar/Vector`. Initialization could be reduced by preallocating all memory outside of the time stepping. However, this would be more difficult to use in the main program.

References

[1] Matteo Frigo. A fast fourier transform compiler. *SIGPLAN Not.*, 34(5):169–180, May 1999.

[2] Brad Whitlock. *Getting Data into Visit.* Lawrence Livermore National Laboratory, 2010.

[3] Christian Bell, Dan Bonachea, Rajesh Nishtala, and Katherine Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 84–84, Washington, DC, USA, 2006. IEEE Computer Society.

[4] Anthony Chan, Pavan Balaji, William Gropp, and Rajeev Thakur. Communication analysis of parallel 3d fft for flat cartesian meshes on large blue gene systems. In *Proceedings of the 15th International Conference on High Performance Computing*, HiPC'08, pages 350–364, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Bachan J, Baden S, Bonachea D, and Hargrove P. UPC++ specification v1.0, draft 6. *Lawrence Berkeley National Laboratory Tech Report*, 2018.

[6] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[7] C. Canuto. *Spectral Methods in Fluid Dynamics.* Springer-Verlag, 1988.