Accelerating Turbulence Computations on Blue Waters

P.K. Yeung[1] and D. Pekurovsky[2]

[1]Schools of Aerospace Engineering, Computational Science and Engineering,
and Mechanical Engineering, Georgia Institute of Technology, Atlanta, GA 30332

[2]San Diego Supercomputer Center,
University of California San Diego, La Jolla, CA 92093

ABSTRACT

Direct numerical simulation of turbulence using Fourier pseudo-spectral methods is a very communication intensive application in high performance computing. Several techniques, including multithreading, remote memory addressing, coarse-grained overlapping based on non-blocking collective communication, and use of dedicated partitions, have been tested in pursuit of improving code performance on Blue Waters at large problem sizes. We discuss the lessons learned and the most promising approaches for sustained production runs in the future.

1 INTRODUCTION

Turbulence is a very common state of fluid motion where the flow is three-dimensional (3D), unsteady, stochastic, and is characterized by nonlinear interactions among fluctuations spanning a wide range of scales, in both length and time. The flow around an airplane and in its jet engines, the dispersion of smoke from a chimney, and disturbances in the atmosphere that give rise to stormy weather, to name a few, are all turbulent. As a result, given the complexity of the subject (see, e.g., [1],[2]), it is not surprising that limitations in understanding turbulence and or manipulating its effects often play a key role in important technological problems of societal concern.

The highest-fidelity approach to computing turbulence is direct numerical simulation (DNS) [3], in which the instantaneous velocity field is computed from the basic governing equations for conservation of mass and momentum. However, to capture all scales reliably the domain size must be large, and the grid spacing must be small; and similar consideration applies to time-stepping as well. Since the range of scales increases rapidly with the Reynolds number (which is high in most applications), it is clear that state-of-the-art simulations are highly resource-intensive and tied to advances in supercomputing power worldwide (e.g. [4, 5, 6, 7]).

Most theories in turbulence are dependent on assumptions in the physics of how different scales interact at high Reynolds number, usually with some degree of scale similarity relatively insensitive to the boundary conditions. In principle, at least, the fundamental physics of small and intermediate scales can be addressed by considering unbounded homogeneous turbulence on a 3D domain with periodic boundary conditions. The velocity field is represented by a number of Fourier modes, and numerical solutions can be obtained using a pseudo-spectral approach (see [8],[9]). Nonlinear terms in the equations of motion are formed by straightforward multiplication in physical space instead of using (a much more costly) convolution integral in wavenumber space, provided suitable measures are taken to mitigate the aliasing errors that arise. The primary mathematical operation is the three 3D Fast Fourier Transform (FFT), which has an operations count scaling as $N^3 \ln_2 N$, and has many other applications in science and engineering.

To allow the code to scale up to very core counts, we use a 2D domain decomposition [10, 11], in which the data are divided in two directions and stored as a collection of *pencils* in the local memory of a large number of parallel MPI processes. This decomposition also defines a 2D MPI Cartesian processor grid, of dimensions $M_1 \times M_2 = M$ where M is the number of MPI *tasks*. However, because Fourier series are global in nature, completion of a full 3D transform requires data to be transposed between pencils aligned with different coordinates axes, at each time step of the DNS code. This transpose requires message passing of a collective nature, among all MPI tasks belonging to one of M_2 row communicator (each of size M_1) or one of M_1 column communicators (each of size M_2). Communication costs are the primary bottleneck, especially at large core counts where communication overhead may increase significantly. On a large shared system like Blue Waters, network contention for communication bandwidth with other jobs on the system

also causes significant variability.

The considerations above lead to a strong motivation to reduce net communication costs as the primary route improving code performance. Accordingly the overall objective of our activities within the terms of the Blue Waters NEIS-P2 subaward is to explore the use of various alternative programming models, including multithreading, remote memory addressing, and other approaches that became feasible as our efforts progressed. Most of the work required to test and evaluate a new programming model or a new approach to perform communication is carried out using a 3D FFT kernel that has a data structure similar to the DNS code, which we refer to below as PSDNS (Petascale DNS).

Currently our production PSDNS code is a hybridized MPI-OpenMP version, written in the so-called "funneled" mode where only the master thread on each MPI task makes communication calls. The code performs as pure MPI if the number of threads per MPI task is unity. Collective communication required in the data transposes is performed in a blocking manner, using either standard MPI_ALLTOALL (or MPI_ALLTOALLV) functions or Fortran co-arrays in the Cray Compiler Environment. Use of co-arrays, which is based on remote memory addressing principles, usually gives better performance especially at large problem sizes. We have used FFT kernels to study possible benefits from the use of several other programming models, such as overlapping between master-thread communication and worker-thread computation using OpenMP, overlapping between computation and communication using non-blocking collective communication, as well as the use of graphical processor units via the OpenACC accelerator library. The codes have also been tested on other leading-edge platforms at multiple sites. However in this document we focus on developments on Blue Waters, including machine-dependent considerations such as the use of reserved and dedicated partitions.

The rest of the paper is organized as follows. In the next section we give the mathematical formulation of the equations solved in the DNS code. In Sec. 3 we discuss the formulation and performance of the parallel algorithm we currently use for production purposes. In Sec. 4 we report on several strategies that we have tested in pursuit of performance improvement on Blue Waters. It should be noted that further efforts are to be made, especially along the lines described in Secs. 4.3. The best performance to-date, however, has been obtained when running on a dedicated partition as reported in Sec. 5. We conclude in Sec. 6 with a summary of the lessons learned and a further discussion of the possible use of dedicated partitions for sustained production runs in the future.

2 Mathematical Formulation

The PSDNS code computes instantaneous velocity fields $\mathbf{u}(\mathbf{x}, t)$ according to the Navier Stokes equations which represent conservation of momentum, in incompressible flow where the velocity vector is solenoidal ($\nabla \cdot \mathbf{u} = 0$). Assuming zero mean velocity, we can write

$$\partial \mathbf{u}/\partial t + \mathbf{u} \cdot \nabla \mathbf{u} = -\nabla(p/\rho) + \nu \nabla^2 \mathbf{u} + \mathbf{f}, \qquad (1)$$

where the density (ρ) and viscosity (ν) are constant, and $\mathbf{f}$ denotes any body forces such as those representing solid body rotation or magnetohydrodynamic effects, as well as numerical forcing which is often used to maintain the energy of the velocity fluctuations against viscous decay. Important phenomena in turbulent mixing and dispersion can be studied by solving additional equations for passive scalar field, and by following the trajectories of fluid or suspended articles whose velocities may be obtained by interpolation from velocities at neighboring grid points.

By transforming Eq. 1 to Fourier space one can readily show that velocity Fourier coefficient $\hat{\mathbf{u}}(\mathbf{k}, t)$ evolves by

$$(\partial/\partial t + \nu k^2)\hat{\mathbf{u}} = \hat{\mathbf{C}}_{\perp \mathbf{k}} \qquad (2)$$

where on the r.h.s. the nonlinear convective terms are projected onto the plane perpendicular to $\mathbf{k}$ in Fourier space. For each $\mathbf{k}$, Eq. 2 can be treated as an ordinary differential equation in time, for which explicit Runge-Kutta methods are applicable. We use a a variant of Eq. 2, namely in the formulation of [12], which allows a slight reduction of the number of variables that must be transformed at each time step. Aliasing errors are controlled by truncation and phase-shifting techniques as described in [12]. Because the velocity field is real-valued, conjugate symmetry applies, i.e. $\hat{\mathbf{u}}(-\mathbf{k}) = \hat{\mathbf{u}}^*(\mathbf{k})$. This also means that for every N^3 grid points, only $\frac{1}{2}N^3$ Fourier modes are independent. Accordingly, Fourier modes with negative wavenumbers in one arbitrarily chosen (x) direction are not stored explicitly in the code.

At the beginning of each Runge-Kutta substep the velocity field is in Fourier space, with each MPI task holding pencils of data aligned in the y direction. We first take a complex-to-complex (C-C) FFT to physical space in the y direction. The data then needs to be transposed into pencils aligned in z, whereupon a C-C FFT is performed in z. Finally the data are transposed into pencils aligned in x, and (because only modes with $k_x \geq 0$ are kept) a complex-to-real (C-R) transform is taken. All variables are then in physical space, where nonlinear terms are subsequently formed by multiplication. A reverse cycle of transforms and transposes back to Fourier space gives the r.h.s of Eq. 2 (or its equivalent). Finally explicit time-stepping according to the chosen Runge-Kutta

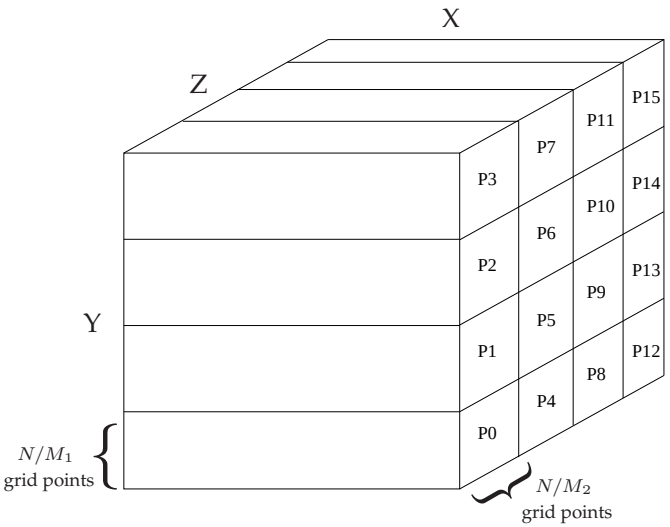

Figure 1: Illustration of 2D domain decomposition with $M_1 = M_2 = 4$ ($M = 16$).

scheme is applied, followed by a repeat for subsequent Runge-Kutta substeps that constitute together a complete time step in the code.

As noted earlier we use an FFT kernel to assess the effectiveness of new programming models. The action of the kernel consists of setting up a simple sinusoidal fields in physical space, such as $u = \sin(x)\cos(y)\sin(z)$, transforming it to wavenumber space, and back, for a number of times. Elapsed wall times taken in major subroutines are recorded and also broken down into costs incurred in communication, FFTs (in each direction), and the packing and unpacking which are necessary because communication calls must be executed using contiguous messages. Correctness of results is checked by comparing the spectrum to the expected result of a single spike, and by requiring the maximum absolute difference between initial and final results to be below thresholds for round-off errors depending on the use of single versus double machine precision. Because the numerical algorithm we use ([12]) includes operations carried out on partially transformed variables at the immediate stages, in the DNS code we do not directly use a P3DFFT library developed by one of the authors [11]. However, many principles are similar, and newly developed versions of key routines in the FFT kernel can be used directly in the DNS code without significant changes.

3 Base algorithm and code performance

3.1 Current algorithms

We consider the parallel algorithm in more detail here, mainly from the perspective of the FFT kernel, which (unlike the DNS code at the beginning of a time step) is designed to begin with pencils aligned with x in physical space, based on a 2D domain decomposition as illustrated in Fig. 1. Each MPI task belongs to a row communicator as well as a column communicator. For efficient arithmetic, the array structure is designed to allow FFTs to be taken with vector stride unity one direction at a time. The sequence of operations starting from having data in physical space and x-pencils is laid out in Table 1.

1. R-C FFT in x direction.
2. Pack data into contiguous messages
3. Alltoall exchange within row communicator.
4. Unpack receiving buffer into z pencils.
5. C-C FFT in z direction.
6. Pack data into contiguous messages
7. Alltoall exchange within column communicator.
8. Unpack receiving buffer into y pencils.
9. C-C FFT in y direction.

Table 1: Sequence of operations for 3D FFT from physical space to wavenumber space.

Operations 1-9 above together constitute a complete 3D FFT from physical space to wavenumber space, which we denote by *xktran*. Operations 1-4 and 6-9 are collected into two subroutines, called *xkcomm1* and *xkcomm2* respectively. Generally, we find better performance by using a Cartesian processor grid with M_1 small compared to M_2 and equal to the number of cores per node, primarily because such an arrangement allows communications within a row communicator to occur entirely on node. (On Blue Waters, this means letting M_1 be 16 in single stream mode, 32 in dual-stream mode.) For the same reason, *xkcomm2* is more time-consuming than *xkcomm1*. It is well understood that alltoall exchanges (especially on a column communicator) pose the greatest constraint on scalability. However, depending on size of cache and memory bandwidth, the time spent on packing and unpacking, also referred to as "local transpose" operations, can be substantial as well. An inverse transform involves a reverse sequence of operations. The times taken for each half of a forward-backward transform pair are generally comparable.

Many factors influence the elapsed wall time per FFT transform cycle (in the FFT kernel) or per time step (in the DNS code). For the time spent on communication, at a high level we can consider the total volume of message traffic, the number and size of individual messages (with total volume fixed), the manner in which the messages are sent and received, and the effects of machine topology and hardware and software environment.

Message volume. Clearly, the total volume of the message traffic depends on the number of grid points (N^3), machine precision (4 bytes per word assuming single precision), and the number of variables that need to be

transformed and transposed as determined by the logic in the DNS code. In [12]'s formulation this number is $n_v = 3 + 2\ max(1, n_\phi)$, where $0 \le n_\phi \le 2$ is the number of so-called passive scalars (if any) carried by the velocity field. However, we are able to exploit a particular feature of the dealiasing error treatments in the code to reduce both message traffic and computational workload significantly. Specifically, in order to remove multidimensional alias errors, Fourier modes with wavenumber magnitude $|\mathbf{k}| > k_{max}$ where $k_{max} = \sqrt{2}N/3$ (assuming a standard $(2\pi)^3$ domain) are truncated (set to zero). Fourier coefficients for such truncated modes do not need to be transposed or transformed. Within our 2D domain decomposition approach it is convenient explicitly skip modes that fall outside a cylinder of radius k_{max} in the $k_x - k_z$ plane, under the condition

$$(k_x^2 + k_z^2)^{1/2} > \sqrt{2}N/3\ . \tag{3}$$

We refer to this scheme as "cylindrical truncation". Although this captures only a subset of modes with $|\mathbf{k}| > k_{max}$ the fraction of modes explicitly dropped (proportional to the space between the boundaries of a square of length $N/2$ and a circumscribed circle of radius $\sqrt{2}N/3$) is still quite significant. In many cases the savings in cost is of order 30%, provided care is taken to manage the resulting imbalance of workload among different MPI tasks, since each task then works on a different number of non-truncated modes.

Message size. For a fixed volume of message traffic, message size depends on the number of MPI tasks, the processor grid geometry, and whether multiple variables are transposed separately or together. When communication is performed using MPI, we find that transposing multiple variables together (which leads to fewer but larger messages) usually brings a slight improvement (in the order 5-10%), since the overhead due to latency is reduced. However, if communication is performed using Fortran co-arrays (see below) better performance is obtained by explicitly dividing messages into smaller chunks. The optimal message size may also depend on environmental variables in the job script which can influence whether messages of a given size will be exchanged using the so-called *eager* or *rendezvous* protocols.

OpenMP multithreading. The advent of multi-cored processors has made possible a mode of hybrid programming where MPI tasks handle communication but threads spawned from the MPI tasks perform computation. We have hybridized the production DNS code completely, in the so-called *funneled* mode where only the master threads make communication calls (while other threads wait) and each thread carries out computation on different parts of the data. The code is equivalent to pure MPI if the number of threads per MPI task (*num_thr*) is one. The total number of cores used then becomes the product of number of tasks (M) and *num_thr*. Since penalty for communication overhead increases with the number of tasks, for relatively large problem sizes an increase of *num_thr* with M fixed may give better scaling than an increase of M with *num_thr* fixed (effectively, at unity). However, although *num_thr* can in principle range from 1 to the number of cores per node, because of NUMA cache coherency considerations a value of 4 or more usually does not perform better than pure MPI.

Fortran Co-Array. A strategy for data exchange fundamentally different from standard MPI message passing is to implement remote direct memory access (RDMA) protocols, whereby each MPI task is given permission to update selected areas of the memory of another MPI task (and vice versa). Conceptually this is analogous to MPI-2 one-sided communication, but the implementation on Cray XE and XK platforms within the Cray Compiler Environment (CCE) is apparently very efficient. We are indebted to Dr. R.A. Fielder and his colleagues at Cray Inc. for providing us with an efficient version of the key *xkcomm1* and *xkcomm2* (and similar) routines using Fortran Co-Arrays. (This approach was formerly referred to as Co-Array Fortran, invoked in compilation by `-h caf`, but recently this has been absorbed into the new Fortran standard.) Essentially, the receiving buffers at operations 4 and 8 listed earlier in Table 1 are declared as co-arrays, and data exchange is accomplished by a remote assignment statement. Unlike the case of communication using standard MPI, smaller messages are preferred when using CA since a different system protocol is used to handle the actual transfers. In routines written by Dr. Fiedler the data to be exchanged are broken down into buckets of 512 bytes. We have found consistently that at large problem sizes and high core counts compared to the CA version of our PSDNS code consistently perform better than the other currently implemented options.

3.2 BENCHMARKING AND BASIC PERFORMANCE DATA

The reliability of performance data used to guide decision-making in our code development processes is important. In each DNS time step, or each iteration in the FFT kernel, the elapsed wall time varies among different MPI tasks, and between one step/iteration and the next. Since all MPI tasks are synchronized from time to time, the slowest MPI task takes a determining role. On the other side since variability is largely external to the code, it is reasonable to take the most favorable time from several steps (each being from the slowest MPI task). Furthermore, since the user has freedom to specify the processor grid geometry we only report data obtained from the best $M_1 \times M_2$ combination in each case. It is generally

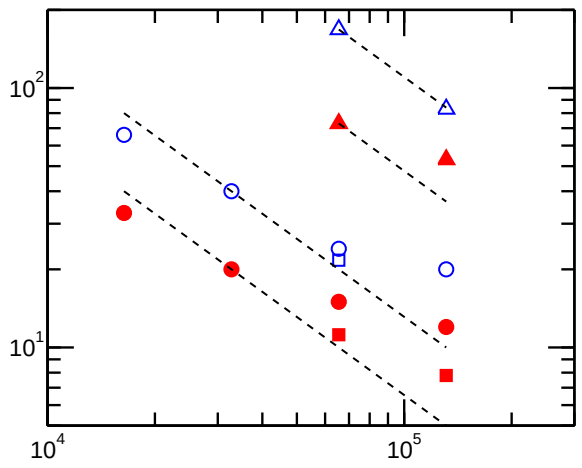

Figure 2: CPU timings per step of production DNS code in single-precision, versus the number of cores used, recorded on 16-cores Cray XK6 (Jaguarpf at Oak Ridge National Laboratorier, summer 2012). the number of cores used. All data points were obtained using pure MPI except the two solid squares wwhich were obtained with num_thr=2 with number of MPI tasks halved. Grid resolutions tested were 4096^3 (circles) and 8192^3 (triangles). Data points in red are for simulations of velocity field only ($n_\phi = 0$), blue are for those with two scalars ($n_\phi = 2$). Dashed lines of slope 1 indicate limits of perfect scalabilty. Fourth-order Runge Kutta scheme was used; timings for second-order are half of the numbers shown.

fair to compare MPI and MPI-OpenMP codes at the same number of cores used. If possible, if the intention to perform experiments using different approaches at a fixed core count, we try to reduce the impact of variability as a factor in the comparisons by running multiple cases in different sub-directories from the same job script. (This ensures the same nodes on the system are used in each case.)

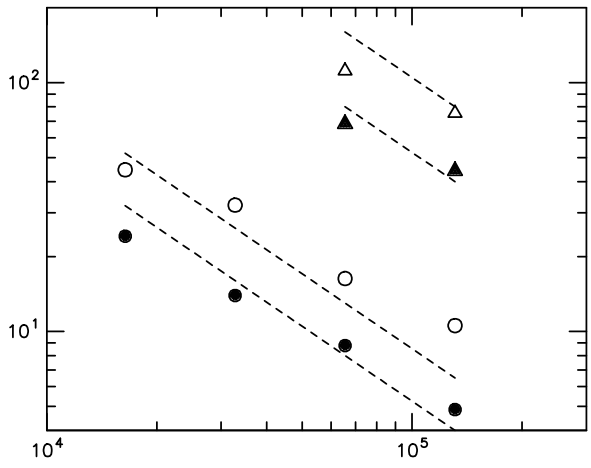

Figure 3: Timings corresponding to those in Fig. 2, but obtained using Co-Array Fortran for alltoall data exchanges.

Figures 2 and 3 show results on CPU timings for the DNS code, on the Oak Ridge Cray XK6 which had 16 cores per node. Although the machine configuration tested then was different from Blue Waters, we expect these data points to provide at least good initial guidance for what can be achieved on Blue Waters. It can be seen in Fig. 2 that scalability for the pure MPI version begins to trail off as number of cores reaches 65536 or higher, but OpenMP produced significant improvements at this core count for the 4096^3 problem. At the same time, the timings from CAF shown in Fig. 3 are consistently are favorable to those in Fig. 2 in all cases represented in both figures. An improvement in strong scaling (deviating less from the ideal slope of -1.0) is also evident.

4 New Programming Models

Here we report on our recent work in pursuit of three new programming models aimed at performance improvement. Overall these strategies have not been as successful as we would like. although in some cases current results may not yet be final. Still, there have been useful exercises, as summarized below.

4.1 Overlap using OpenMP

OpenMP provides for three possible levels of thread safety, known as *funneled*, *serialized*, and *multiple* respectively. In *funneled* mode when the master thread performs communication the other threads are idle. Some overlapping is possible if at any one time some threads are communicating while others are computing. However care must be taken to coordinate traffic among the threads, so that the network bandwidth can be utilized to maximum effect with little idle time, without leading to conflicts known as race conditions (which will cause incorrect results). In *serialized* mode an ORDERED OpenMP construct is used to enforce a strict ordering among the threads. In *multiple* mode all threads can make MPI calls, with no restrictions,

A pipelined procedure has been implemented in a serialized MPI-OpenMP version of the FFT kernel. Suppose we use 4 threads per MPI task, and let each thread undertake operations 1-4 listed in Table 1, and beyond. To begin, all threads can carry out FFT (R-C in the x direction) simultaneously. Then thread 0 engages in alltoall exchange (with the other threads 0 on other MPI tasks). When this done, thread 1 performs the alltoall as thread 0 proceeds to the unpacking, simultaneously. The sequence then repeats until all threads have had their turns at the alltoall. To ensure correct results some penalty is incurred for explicit synchronization. Furthermore if time taken in unpacking (operation 4) is not close to that in alltoall (operation 3) then any advantage gained by overlapping will be quite limited. In any case, we have extended

N^3	Cores	Tasks_Threads	CPU(secs, F/S/M)
2048^3	4096	$16 \times 128\_2$	3.65 / 3.03 / 2.99
2048^3	4096	$8 \times 128\_4$	5.20 / 4.27 / 3.74
4096^3	32768	$16 \times 1024\_2$	6.15 / 6.11 / 6.74
4096^3	32768	$8 \times 1024\_4$	6.75 / 6.58 / 7.40

Table 2: FFT kernel performance data using OpenMP, on Blue Waters, for funneled (F), serialized (S), and multiple (M) modes.

1. FFT and pack, variable #1
2. mpix_ialltoall (req), variable #1
3. loop over remaining variables: j=2,nv
   (a) FFT and pack variable j
   (b) wait (on 'req') for variable j-1
   (c) mpix_ialltoall (req), variable j
   (d) unpack, variable j-1
4. wait and unpack for last variable

Table 3: Pseudo-code for overlap using non-blocking collectives. Each non-blocking communication request is given a tag, denoted by 'req' here for short.

this pipelined procedure to *multiple* mode as well, where all threads are allowed to perform alltoall independently, with no explicit synchronization, but at the risk of inefficiency due to heavy traffic.

Some OpenMP FFT kernel performance data comparing different levels of thread safety as discussed above are given in Table 2. For comparison, pure MPI for the same number of cores gives 3.20 secs for 2048^3 (4096 cores) and 5.15 secs for 4096^3 (32768 cores). The relative merits of the three thread-safety modes are unclear, both among themselves and versus pure MPI at large core counts. Significant variability between successive trials is also seen.

4.2 Overlap via non-blocking MPI collectives

The appeal of non-blocking communication calls for allowing overlap with computation (while the messages are in transit) is well known, but not commonly well demonstrated in practice. Our effort is a recent one, made possible by the availability of non-blocking MPI alltoall (mpix_ialltoall, with a request handle as additional argument) on Blue Waters since March 2013.

We have implemented overlapping by non-blocking MPI alltoall in a coarse-grained fashion, considering work needed for a 3D FFT of *nv* variables. Since alltoall's on column communicators are of greater concern we illustrate in Table 3 the logic in our overlapping version of *xkcomm2* that carries out operations 6-9 of Table 1. In Table 4 we compare *xkcomm2* timings for 4 cases, namely (i) standard (blocking) co-arrays, (ii) standard (blocking) MPI, (iii) non-blocking MPI followed immediately by MPI_WAIT, and (iv) non-blocking MPI with overlap as described.

Grid points	1024^3	2048^3	4096^3
No. of MPI tasks	256	2048	16384
Blocking Co-Arrays	0.862	4.06	4.01
Blocking MPI	0.728	3.98	3.17
Non-blocking MPI	0.550	3.91	4.97
Non-blocking MPI w/ Overlap	0.562	3.33	4.88

Table 4: FFT kernel *xkcomm2* timings for methods (i) to (iv) as noted in the text (Sec. 4.2). Number of variables (nv) is 5.

The trends suggested by the present timing data are not well understood (and may have been contaminated by some spurious degree of network contention). However, considering other routines (such as *xkcomm1*, *kxcomm1*) the general picture seems to be non-blocking MPI communication is mainly advantageous at small problems sizes and core counts but does not win at large problem sizes and core counts. One likely reason may be that when running at large scales CPU times spent on FFT and packing/unpacking are only a small fraction of time taken by communication (which inherently scales with core count less well). In other words, the potential of gains from overlapping at these scales may be limited.

4.3 Overlap using Co-Array Fortran

The logic for overlapping as laid out in Table 3 can be readily extended to Co-Array Fortran, provided the remote addressing data exchange statements can be implemented in a non-blocking manner. This is done using a special directive `!dir$ pgas defer_sync` followed by a `sync memory` at an appropriate place. We have had an opportunity to test this on dedicated boxes (see Sec. 5) at the largest problem size of interest, namely 8192^3 resolution on 8192 XE nodes. However, at 5 variables, the non-blocking CAF code with overlap takes 12.6 secs for its *xkcomm2* operations, which is longer than 9.0 seconds using blocking CAF. Thus a preliminary conclusion is that no clear benefits from overlap are obtained at this scale.

It is possible that the ratio between computation and communication (within a 3D FFT) is inherently unfavorable to overlapping at the largest scales. However when using CAF for data exchange part of the cost is spent on copying between a regular array and a co-array. It will be interesting to see, with substantial further effort, whether this copying for one variable can be overlapped effectively with data movement for another variable.

5 Network contention and dedicated boxes

As mentioned in Sec. 1, we have experienced significant performance variability due to network contention on Blue Waters. This variability is generally more serious at larger problem sizes and larger core counts. For example in a series of production jobs at 8192^3 resolution on 4k XE nodes using 16 cores per node, it was not uncommon to see 100% variability, i.e. a time step may take about 25-30 secs to complete in one run but 50 seconds or more in another. The magnitude of this variability is comparable or greater than observed differences between different implementations (such as pure MPI versus Co-Arrays), therefore making comparisons between different implementations difficult at this scale. As noted in Sec. 3.2, in order to draw safer conclusions in assessing the effects of various performance-enhancement features in our codes we generally run different implementations from the same job script, which at least ensures that comparisons are based on the same network topology of nodes running the code. However, even with these safeguards, significant slowdowns or penalties can still arise if nodes assigned to the user job are physically far apart within the Blue Waters node network, or as demand on bandwidth from jobs by other users happen to fluctuate strongly within the duration of our job. The impact of these contention issues is also consistently traceable to the communication costs according to a detailed analysis of the benchmarking data that have obtained.

The best cure to the contention issues is apparently to run on a dedicated partition of the system. This should minimize (if not remove) contention with other jobs of the system, and a partition can be configured with a topology that also likely leads to the fastest communication possible. By special assistance of the Blue Waters system administrators we were able to test (in June 2013) the production DNS code at 8192^3 on 8192 XE nodes, arranged in a *sheet* topology. We used single precision, with a 32×8192 processor grid, second-order Runge Kutta scheme, and the blocking Co-Array routines for communication. The elapsed wall time per step (based on the slowest MPI task) was 15.4 secs for velocity field only, and 26.9 secs with two scalars. These timings compare very favorably with the *best* of all non-dedicated runs conducted using the same parameters, which were 20.5 and 58.1 secs respectively. The impact of dedicated partitions is even greater when one considers that the performance of non-dedicated runs often vary upwards significantly: e.g. getting 35 secs and more instead of 20.5 secs.

6 Summary and Outlook

In this document we have reported on the use of several strategies to improve the performance of a major pseudo-spectral turbulence simulation code on Blue Waters. The algorithmic requirements are closely tied to three-dimensional Fast Fourier Transforms, which are useful in various other fields in science and engineering. More importantly, the lessons learned are relevant to communication-intensive domain science codes running on large shared systems such as Blue Waters.

The methods we pursued perhaps can be generally classified as those which allows the code to (i) perform less communication or communicate more efficiently, (ii) overlap communication with some computation, and (iii) minimize variability and performance degradation caused by network contention. For (i), cylindrical truncation exploiting aliasing error treatment and Fortran Co-Arrays in the Cray Compiler Environment using remote memory addressing have been found to be consistently effective, the latter especially so for large problems. For (ii), we have tested FFT kernels based on overlapping using OpenMP multithreading, non-blocking MPI collectives, and non-blocking Co-Arrays, but have not been able to demonstrate clear improvement. A likely reason is that the ratio of times spent in computation to communication is low at large core counts. However further progress using Co-Arrays may be possible via further enhancements of the techniques currently used. For (iii), tests at the size of our target science problem, namely 8192^3 grid points show that dedicated partitions have, as expected, a highly favorable impact on the timings.

Together with avoidance of random performance degradation regularly observed in un-dedicated runs, we argue that use of dedicated partitions for the largest targeted science problems is essentially vital for a rapid rate of progress at the largest problem sizes of interest. While frequent use of such partitions by reservation may lead to some scheduling inconvenience for other users, a reduction in the level of mutual interference among jobs of various sizes may also allow more science throughput on the system overall.

Acknowledgments

We gratefully acknowledge support from the National Science Foundation (Office of Cyberinfrastructure and the Fluid Dynamics Program). In particular, this work is made possible through a PRAC award from NSF and a NEIS-P2 subaward through the University of Illinois at Urbana-Champaign. We are grateful to Dr. R.A Fiedler of Cray Inc. for providing us with Co-Array Fortran routines and valuable advice on many other fronts. We also

thank several Blue Waters staff members (listed alphabetically: G.H. Bauer, T.A. Cortese, S. Islam, J. Kim, J. Li) for their help in the work reported in this document. Some of the work was performed by PhD students K. Iyer and D. Buaria at Georgia Tech.

REFERENCES

[1] K. R. Sreenivasan. Fluid turbulence. *Rev. Mod. Phys.*, 71:s383–s395, 1999.

[2] J. L. Lumley and A. M. Yaglom. A century of turbulence. *Flow Turb. Combust.*, 66:241–286, 2001.

[3] P. Moin and K. Mahesh. Direct numerical simulation: A tool in turbulence research. *Annu. Rev. Fluid Mech.*, 30:539–578, 1998.

[4] M. Yokokawa, T. Itakura, A. Uno, T. Ishihara, and Y. Kaneda. 16.4-tflops direct numerical simulation of turbulence by a fourier spectral method on the earth simulator. In *Proceedings of the Supercomputing Conference*, Baltimore, November 2002.

[5] J. Jiménez. Computing high-Reynolds-number turbulence: will simulations ever replace experiments? *J. Turb.*, 4:022, 2003.

[6] T. Ishihara, T. Gotoh, and Y. Kaneda. Study of high-Reynolds number isotropic turbulence by direct numerical simulation. *Annu. Rev. Fluid Mech.*, 41:165–180, 2009.

[7] P. K. Yeung, D. A. Donzis, and K. R. Sreenivasan. Dissipation, enstrophy and pressure statistics in turbulence simulations at high reynolds numbers. *J. Fluid Mech.*, 700:5–15, 2012.

[8] S. A. Orszag. Numerical simulation of incompressible flows with simple boundaries. 1. galerkin (spectral) representations. *Stud. Appl. Math.*, 50:293–xxx, 1971.

[9] C. Canuto, M. Hussaini, A. Qarteroni, and T. Zang. *Sectral Methods in Fluid Dynamics*. Springer, 1988.

[10] D. A. Donzis, P. K. Yeung, and D. Pekurovsky. Turbulence simulations on $o(10^4)$ processors. In *TeraGrid 2008 Conference*, Las Vegas, June 2008.

[11] D. Pekurovsky. P3dfft: A framework for parallel computations of fourier transforms in three dimensions. *Siam J. of Scientific Computing*, 34:C192–209, 2012.

[12] R. S. Rogallo. Numerical experiments in homogeneous turbulence. NASA TM 81315, NASA Ames Research Center, Moffett Field, CA., 1981.