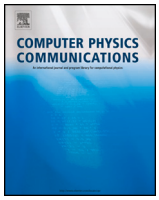
Computer Physics Communications 306 (2025) 109364

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc



Computational Physics

GPU-enabled extreme-scale turbulence simulations: Fourier pseudo-spectral algorithms at the exascale using OpenMP offloading ☆,☆☆

P.K. Yeung [a,b,*], Kiran Ravikumar [a,1], Stephen Nichols [c], Rohini Uma-Vaideswaran [a]

[a] School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA
[b] School of Mechanical Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA
[c] Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

ARTICLE INFO

Keywords:
Exascale
Turbulence
Direct numerical simulations
OpenMP offloading
3D fast Fourier transform
GPU-aware MPI

ABSTRACT

Fourier pseudo-spectral methods for nonlinear partial differential equations are of wide interest in many areas of advanced computational science, including direct numerical simulation of three-dimensional (3-D) turbulence governed by the Navier-Stokes equations in fluid dynamics. This paper presents a new capability for simulating turbulence at a new record resolution up to 35 trillion grid points, on the world's first exascale computer, *Frontier*, comprising AMD MI250x GPUs with HPE's Slingshot interconnect and operated by the US Department of Energy's Oak Ridge Leadership Computing Facility (OLCF). Key programming strategies designed to take maximum advantage of the machine architecture involve performing almost all computations on the GPU which has the same memory capacity as the CPU, performing all-to-all communication among sets of parallel processes directly on the GPU, and targeting GPUs efficiently using OpenMP offloading for intensive number-crunching including 1-D Fast Fourier Transforms (FFT) performed using AMD ROCm library calls. With 99% of computing power on Frontier being on the GPU, leaving the CPU idle leads to a net performance gain via avoiding the overhead of data movement between host and device except when needed for some I/O purposes. Memory footprint including the size of communication buffers for MPI_ALLTOALL is managed carefully to maximize the largest problem size possible for a given node count.

Detailed performance data including separate contributions from different categories of operations to the elapsed wall time per step are reported for five grid resolutions, from 2048^3 on a single node to 32768^3 on 4096 or 8192 nodes out of 9408 on the system. Both 1D and 2D domain decompositions which divide a 3D periodic domain into slabs and pencils respectively are implemented. The present code suite (labeled by the acronym GESTS, GPUs for Extreme Scale Turbulence Simulations) achieves a figure of merit (in grid points per second) exceeding goals set in the Center for Accelerated Application Readiness (CAAR) program for Frontier. The performance attained is highly favorable in both weak scaling and strong scaling, with notable departures only for 2048^3 where communication is entirely intra-node, and for 32768^3, where a challenge due to small message sizes does arise. Communication performance is addressed further using a lightweight test code that performs all-to-all communication in a manner matching the full turbulence simulation code. Performance at large problem sizes is affected by both small message size due to high node counts as well as dragonfly network topology features on the machine, but is consistent with official expectations of sustained performance on Frontier. Overall, although not perfect, the scalability achieved at the extreme problem size of 32768^3 (and up to 8192 nodes — which corresponds to hardware rated at just under 1 exaflop/sec of theoretical peak computational performance) is arguably better than the scalability observed using prior state-of-the-art algorithms on Frontier's predecessor machine (*Summit*) at OLCF. New science results for the study of intermittency in turbulence enabled by this code and its extensions are to be reported separately in the near future.

☆ This manuscript has been authored in part by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The publisher acknowledges the US government license to provide public access under the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

☆☆ The review of this paper was arranged by Prof. David W. Walker.

* Corresponding author at: School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, GA 30332, USA.
E-mail addresses: pk.yeung@ae.gatech.edu (P.K. Yeung), kiran.ravikumar@nasa.gov (K. Ravikumar).
[1] Current address: Science and Technology Corporation, NASA Ames Research Center, Moffett Field, CA 94035.

https://doi.org/10.1016/j.cpc.2024.109364
Received 30 April 2024; Received in revised form 1 August 2024; Accepted 2 September 2024
Available online 5 September 2024

0010-4655/© 2024 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

1. Introduction

Fluid turbulence with irregular fluctuations spanning a wide range of scales in time and three-dimensional space is both a domain science problem of great complexity [1–3] and a lasting grand challenge problem in leadership-class computing, even as technology has advanced dramatically through the years [4–6]. The stochastic nature of turbulence provides strong motivation for pursuing a fundamental understanding of the nature of the fluctuations, including the typical intensity and likelihood of extreme events, which can have severe consequences in engineering applications and natural phenomena. The computational approach that is most helpful for physical understanding is direct numerical simulation (DNS [7]), via the numerical solution of exact governing equations for conservation of mass and momentum for the velocity fluctuations. However, DNS is inherently resource-intensive, especially when high resolution and adequate sampling in both time and space are necessary for accurate results. At the same time, advances in computing power since the beginning of this century have enabled a long list of simulation efforts with increasingly high resolution, including Refs. 8–14 and others, if just counting incompressible and non-reacting flows alone.

In this paper, we report on the latest state-of-the-art developments, at a resolution up to 35 trillion grid points, on the world's first exascale computer, named *Frontier* [15–17] at the Oak Ridge Leadership Computing Facility (OLCF), comprising AMD CPUs and GPUs linked by HPE's Slingshot interconnect in a dragonfly network topology. It should be noted that a simulation at 32768^3 resolution has 5.6x and 8x more grid points than prior records at 18432^3 [14] and 16384^3 [12]. This level of grid resolution is expected to provide excellent opportunities to pursue fundamental advances in turbulence research previously limited by the computing powers available, such as by resolving the small scales more accurately, representing a wider range of scales, or meeting additional resource requirements when the turbulence is coupled with mixing, chemical reaction, particulate flows, or magnetic fields.

Given the demands for accuracy and resources, it is not surprising that DNS is usually applied only to flows in simplified, or canonical geometries amenable to high-fidelity yet efficient numerical methods. The best-known example is perhaps the use of orthogonal Fourier representations which are, for a given count of grid points, demonstrably more accurate than finite difference methods of any order [18]. The solution domain is periodic in at least one Cartesian coordinate direction, but often in all three, giving essentially a periodic cube. Although highly simplified, this type of domain is, in view of the classical hypotheses of Kolmogorov [19,20], actually ideal for studying aspects of the small scales of turbulence, with results also relevant (via the hypothesis of local isotropy) to more practical geometries provided the Reynolds number is sufficiently high, as is the case in most practical applications.

The principal mathematical operation in our work is the 3D discrete Fast Fourier Transform (FFT) [21], which is itself an active subject of high-performance computing research [22–25] relevant to many other scientific disciplines such as materials science and medical imaging [26,27]. A 3D Fourier representation for any real-valued flow variable $f(\mathbf{x})$ with Fourier coefficients $\hat{f}(\mathbf{k})$ can be written as

$$f(\mathbf{x}) = \sum_{\mathbf{k}} \hat{f}(\mathbf{k}) \exp(i\mathbf{k}\cdot\mathbf{x}) \tag{1}$$

where $i = \sqrt{-1}$, symbols in boldface represent vectors throughout the paper, $\mathbf{x}$ and $\mathbf{k}$ denote position and wavenumber vectors, and summation is taken over all Fourier modes represented on an N^3 grid. Usually, the domain size is chosen as 2π units in each direction, which allow the components of the wavenumber vectors to take integer scales spanning from $1-N/2, 2-N/2, \ldots -1, 0, 1, \ldots N/2$. The reciprocal relation $\hat{f}(\mathbf{k}) = \sum_{\mathbf{x}} f(\mathbf{x}) \exp(-i\mathbf{k}\cdot\mathbf{x})$ with summation over all grid points is also well known. The essence of pseudo-spectral methods is that Fourier transforms of nonlinear terms are obtained by multiplication in physical space and subsequent transformation to wavenumber space, rather than the much more costly convolution sums in fully-spectral approaches. As a result, the success of high-resolution pseudo-spectral DNS on massively parallel computers depends critically on the performance and scalability of 3D FFTs — which for us also serves as the fundamental testbed for any new programming models which may be required for efficient use of a future machine. The need to alleviate high costs of data movement is also a feature shared by high-order finite difference approaches for multi-dimensional fluid flow problems, motivating somewhat unconventional techniques in some cases [28].

In the context of turbulence research, our primary objectives are to perform a simulation as large as the memory on the most powerful system available to us allows, and to produce a code that scales with both problem size and node count as efficiently as possible. The memory-intensive nature of DNS along with the multi-dimensional nature of the Navier-Stokes equations imply that the solution domain must be subdivided among a certain number (P) of parallel processes that can operate on a subset of the data locally on their own, but must also communicate with each other as required by the algorithm. In particular 3D FFTs are performed one direction at a time using highly optimized software libraries if all data points along complete lines of data are available in the local memory. For a complete FFT in three directions, the solution domain must be transposed between partitions along different directions at different stages of the calculation. It is very important that this transpose, which requires collective inter-process communication of the all-to-all variety, is performed efficiently and with as little memory overhead as possible. Indeed, in general, the cost of 3D FFTs, and the turbulence simulation itself, is dominated by data movement, rather than purely computational number crunching. This is also increasingly true at large problem sizes and node counts, leading to limitations on scalability typical of communication-intensive codes built upon the principles of distributed-memory parallelism.

In recent years, the advent of Graphical Processing Units (GPUs) as a form of heterogeneous accelerators, especially with associated advances in memory and communication capabilities, has given new optimism for high-resolution DNS. The three most recent US-based computers that received No. 1 top-500 [29] ranking have all been machines where most of the computing power lies in the GPUs tied to a relatively modest number of large-memory CPU nodes. Each GPU is attached (as a "device") to one or more CPU cores (as "host"). The GPU provides a high degree of parallelism by spawning a large number (typically thousands) of independent execution threads. However, new challenges can arise, in the data movement between the host and device, and in whether communication can be handled efficiently by the GPUs. For instance, the best use of Summit, an IBM-NVIDIA machine at OLCF which allowed us to reach 18432^3 resolution involved a batched asynchronism approach [6] since the GPU memory could not hold the entire problem, and we needed to solve the problem in batches. This approach [6] was very effective in minimizing the overhead of both the batched data movement and the communication by overlapping the computation, data movement, and host-based communication as much as possible. However, with 4x the device memory of Summit, Frontier has parity in the host and device memories, and when combined with an efficient GPU-aware communication capability through HPE Slingshot, we can employ a more straightforward strategy where the complete problem always resides in device memory and explicit copies between device and host are performed only when performing I/O operations, leading to less overhead compared to Summit.

Our codes are written in Fortran 90/95, with Fortran-C interoperability features that allow us to call GPU libraries provided in C, including the AMD-supplied "ROCm" library that we use for 1D FFTs. For other computational and data-copying operations, the design of Frontier makes it relatively straightforward to target GPUs using OpenMP offloading [30], as per the OpenMP standard [31]. Since the OpenMP standard is vendor-independent, OpenMP is expected to provide a smooth path for portability to the next generation of (multi-exaflop) machines [32] as vendors implement the newer OpenMP features into their compilers.

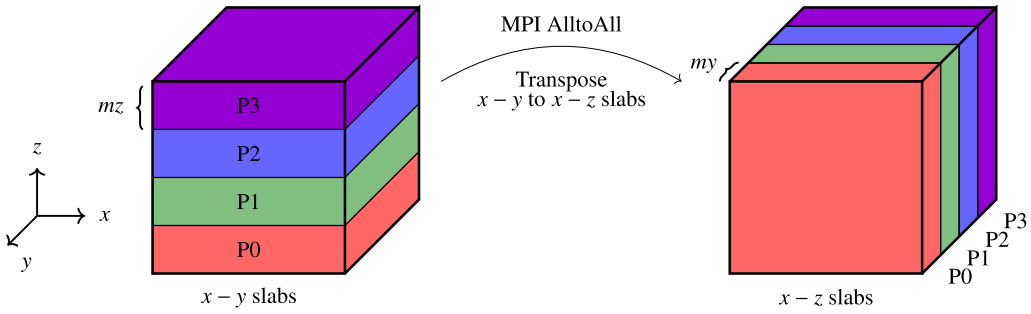


Fig. 1. Schematics of the 1D decomposition showing a cubic domain divided into 4 slabs identified by the color coding and process IDs 0,1,2,3. $my = mz = N/P$ denote the sub-divided range of grid points present in the respective directions in each slab.

In the following sections, we provide a detailed description of the essential algorithmic elements (Secs. 2, 3) and code performance (Secs. 4, 5) currently achieved via two variants of a new GPU-enabled code written specifically for use on Frontier. These codes have been developed under the auspices of the selective Center for Accelerated Application Readiness (CAAR) program on Frontier operated by the OLCF; and are collectively referred to as the GESTS (GPU-enabled Extreme Scale Turbulence Simulations) suite of codes. Code performance data are presented for five problem sizes ranging from 2048^3 which fits on 1 node, to 32768^3 (35 trillion grid points) which requires a minimum of 4096 nodes out of the 9408 available on Frontier. Indicators of high performance include weak scaling from 4096^3 to 32768^3 (with node count in proportion to memory requirements) at nearly 90% or greater, and essentially 100% strong scaling for 16384^3 with a 4x increase in node count from 512 to 2048. In our discussions we also identify remaining challenges which inevitably arise at the extreme limits of current machine capacity, but may potentially benefit from future enhancements in machine hardware or improved exascale-ready software libraries. Our focus here is on simulations of the Eulerian velocity field only. The GESTS suite of codes is also capable of tracking fluid particles and computing the transport of passive scalars, which are both very important aspects of turbulence, but the associated coding developments in those areas will be reported separately. Conclusions are summarized in Sec. 6.

2. 3D FFT and domain decomposition

While our work in this paper is driven by our needs to study turbulence, we provide here a discussion on 3D FFTs, starting from a general perspective that is hopefully of interest to readers in a wider community outside our domain science specialty. Development or porting of a code that performs only 3D FFTs is often our first task when exploring the use of a new platform.

2.1. Distributed-memory parallelism for 3D FFTs

As a "tensor product" operation, a 3D FFT can be performed as three 1D FFTs in each coordinate direction, sequentially. For instance a forward transform from physical space to wavenumber space can be written as:

$$\hat{f}(k_x,k_y,k_z) = \sum_y \exp(-ik_y y)\left\{\sum_z \exp(-ik_z z)\left\{\sum_x \exp(-ik_x x) f(x,y,z)\right\}\right\} \tag{2}$$

where summations cover all grid points (at spacings $2\pi/N$ apart), and a real-to-complex (R-C) transform (in x) is followed by two complex-to-complex (C-C) transforms in z and y. Likewise, for an inverse transform we have

$$f(x,y,z) = \sum_{k_x} \exp(ik_x x)\left\{\sum_{k_z} \exp(ik_z z)\left\{\sum_{k_y} \exp(ik_y y)\hat{f}(k_x,k_y,k_z)\right\}\right\} \tag{3}$$

which indicates two C-C transforms followed by a complex-to-real (C-R) transform. Fourier coefficients of derivatives are readily obtained via multiplication in Fourier space: e.g. $\widehat{\partial f/\partial x} = ik_x\hat{f}$. Since f is real-valued, the property of conjugate symmetry i.e. $\hat{f}(-\mathbf{k}) = [\hat{f}(\mathbf{k})]^*$ (where the superscript $*$ denotes a complex conjugate) holds and implies that Fourier coefficients with negative wavenumber in one specific direction need not be stored. This direction has been chosen as x, which is the first (innermost) transform direction in Eq. (2) but the last (outermost) in Eq. (3).

Domain decomposition enables computations beyond the capacity of a single CPU core by distributingthe work across multiple parallel processes. As illustrated in Figs. 1 and 2, for a 3D FFT, the domain can be divided into up to N *slabs* (each consisting of N/P planes); or up to N^2 *pencils* (each of size equal to $N\times(N/P_r)\times N/P_c$ grid points) where $P_r\times P_c = P$ gives the shape of a 2D (row and column) Cartesian processor grid. Large 3D FFTs are MPI communication-intensive because of the transposes necessary to collect complete lines of data in the local memory before the transforms in the respective directions can be performed. If the memory available to N parallel processes is sufficient for an N^3 problem then the *slabs* approach is preferable, since it requires fewer MPI communication cycles and may give slightly shorter runtimes. However, the *pencils* approach provides greater flexibility where more than N parallel processes are necessary to provide sufficient memory for the problem, or to provide a clear strategy to distribute the problem over a larger set of processes to decrease the runtime.

We consider the basic operations needed for a forward transform, per Eq. (2), in the *slabs* code, starting with the domain divided in the y direction into collections of $x-z$ planes. Per standard practice, so-called (batched)"FFT plans" are constructed to specify the manner in which multiple lines of data can be transformed using a single kernel call. Since array elements are stored in column-major order in Fortran, a real-to-complex FFT in x can be taken immediately, with numbers forming each line of data residing contiguously in the memory. For FFT in z (as the second direction) there are two options. The first is to take a strided transform, where successive elements in z are stored N memory positions apart from each other. The second is to re-arrange the array into $z-x$ planes and then perform a non-strided transform in z. In principle, non-strided transforms are faster because of more efficient memory access, but the overhead of switching $x-z$ planes into $z-x$ planes can be substantial. A similar issue arises when data already transformed in z are packed into a contiguous buffer, passed through a standard MPI_ALLTOALL library call, and then unpacked into intended memory locations as $x-y$ planes before a (strided) C-C FFT in y is performed.

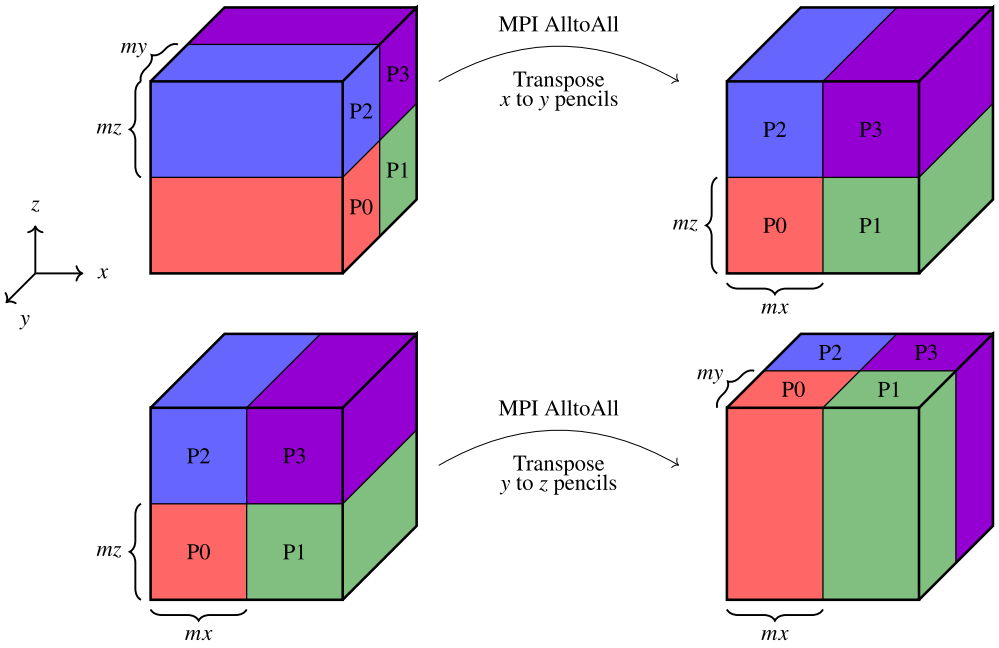


Fig. 2. Schematics of the 2D decomposition of a cubic domain into 4 pencils in a 2×2 processor grid.

Table 1
Elapsed wall time (in seconds) of strided and non-strided complex-to-complex 1D forward FFT for three variables using batched plans.

Size	Strided				Non-strided			
	Unpack	FFT	Pack	Total	Unpack	FFT	Pack	Total
2048^3	0.0368	0.138	0.0336	0.209	0.0789	0.0174	0.0347	0.131
4096^3	0.0340	0.168	0.0336	0.235	0.0879	0.0248	0.0350	0.148
8192^3	0.0340	0.135	0.0336	0.202	0.133	0.0385	0.0873	0.259
16384^3	0.0340	0.221	0.0336	0.288	0.282	0.0366	0.173	0.492
32768^3	0.0340	0.306	0.0336	0.373	0.647	0.0505	0.249	0.947

Table 1 compares the performance of complex-to-complex forward transforms, for three variables, in both strided and non-strided forms using batched FFT plans. It is clear that FFT library calls for non-strided transforms are faster than strided versions. However, the cost of packing and unpacking needed for non-strided transforms is very substantial, resulting in a lower overall cost for using strided transforms at problem sizes 8192^3 or higher. Accordingly, we have used strided transforms in y and z directions in this work.

While in principle any positive integers satisfying $P_r \times P_c = P$ can be used to obtain correct results, it is usually best to set P_r to be equal to or a factor of the number of MPI processes per node, so that communications in the row communicator can run faster by virtue of being on the same node. The two GCDs of each MI250x are connected with each other through a high bandwidth Infinity Fabric interconnect. Therefore, an additional benefit arises if P_r is as small as two, or even one (essentially a slab decomposition). On the other hand, a small P_r leads to a large P_c, which may pose a challenge to scalability at extreme node counts, because of a larger number of smaller messages being exchanged less efficiently over a large number of nodes.

We note here that similar concepts on domain decompositions can also be applied to problems on periodic domains of dimensionality different from three. This is relevant to, e.g. turbulence in two [33] or four [34] dimensions, where (given the same memory available) N is usually larger or smaller than typical values in 3D turbulence, respectively. For 3D problems, the relative merits or utility of domain decompositions of different dimensionalities have been addressed by other authors [22–24,35] as well. As noted already, for an N^3 problem, a slabs code can use only up to N parallel processes. Thus, if N parallel processes cannot provide (i) enough memory for holding all the variables needed for a given calculation, or (ii) sufficiently low wall clock time for given purpose, a pencils code will be required. For instance, if we add particle tracking and/or passive scalar fields then we need a pencils code that can use twice as many nodes. In addition, a 1D decomposition would not be natural if the number of grid points varies between different coordinate directions, which would lead to slabs in different orientations being unequal in size. A 3D block (sometimes called bricks) decomposition does not seem appealing for FFTs, since if complete lines of data are not available in the local memory we can only form (many) partial sums (inside each summation sign in Eq. (2)), and these partial sums will have to be collected together using MPI. Such a scheme will also likely not benefit from highly optimized FFT libraries on the GPU. It is not clear if the increase in code complexity required by a 3D decomposition will pay off. On the other hand, 3D decomposition is useful for visualization, which however does not directly involve FFTs and is not part of regular time steps in our simulations.

2.2. GPU implementation: OpenMP offloading and GPU-aware MPI on frontier

With the successful use of the batched asynchronous algorithm on Summit [6], a natural starting point for code development on Frontier was to adapt the CUDA Fortran implementation of the code using OpenMP to target the AMD MI250x GPUs [36,37]. As mentioned earlier, the significant difference in size between CPU and GPU memory on Summit meant that we could not store the entire problem in the GPU memory and had to perform our computations on the GPUs in

```
1  time1 = MPI_Wtime()
2  if (gpumpi.eq.0) then
3      !$OMP TARGET UPDATE FROM(sndbuf)
4      call MPI_ALLTOALL (sndbuf, nx/2*my*mz, MPI_COMPLEX, &
5          rcvbuf, nx/2*my*mz, MPI_COMPLEX, MPI_COMM_WORLD, mpierr)
6      !$OMP TARGET UPDATE TO(rcvbuf)
7   else
8      !$OMP TARGET DATA USE_DEVICE_PTR(sndbuf, rcvbuf)
9      call MPI_ALLTOALL (sndbuf, nx/2*my*mz, MPI_COMPLEX, &
10         rcvbuf, nx/2*my*mz, MPI_COMPLEX, MPI_COMM_WORLD, mpierr)
11     !$OMP END TARGET DATA
12  end if
13  mpi_runtime = MPI_Wtime() - time1
```

Fig. 3. Code fragment showing the difference between performing MPI_ALLTOALL on host (lines 3-6) vs device (lines 8-11), placed within an OpenMP `TARGET DATA` region, assuming all computations before and after involving the buffer arrays are performed on the device. The communications are wrapped with calls to `MPI_WTIME` to collect the timings.

batches. Moving the batches of data repeatedly between the host and device memory resulted in additional runtime, although that was mitigated to some degree via the asynchronous algorithm [6]. However, with 4x the device memory of Summit, the parity in the CPU and GPU memory on Frontier allows the full problem to reside in the GPU memory and consequently avoids the excessive data movement between the host and device that would be required if solving the problem in batches. Additional improvements in technology provides Frontier with efficient GPU-Aware MPI communications which further permits us to restrict data movement between the host and device to I/O operations only.

With these improvements in mind, it is advantageous and convenient to just copy initial data (often read by the CPU) to the GPU and allow the GPU to handle almost all the work, with only minimal needs for explicit host-device copies. A further simplification is to use a one-to-one correspondence between the CPU cores and the GPU within the same NUMA region on a node. Each Frontier node has four NUMA regions, with each such region consisting of 2 banks of CPU cores and an MI250x GPU which contains two Graphics Compute Dies (GCDs). Each bank of cores is uniquely associated with one GCD, and for optimal use of Frontier's resources, we therefore use 8 MPI ranks per node with 1 CPU core (i.e. "host") and 1 GCD (i.e. "device") per MPI rank.

The OpenMP programming model allows us to enclose almost the entire calculation in a single, large so-called `TARGET DATA REGION`. A `DATA MAP` construct defines the mapping between values of the same variables in the host (i.e. CPU) and device (i.e. GCD) memory. A keyword `TOFROM` instructs the compiler to copy the data to the device at the start of the data region and back to the host at the end of the data region. If and only when necessary, the `UPDATE` clause when combined with `TO` or `FROM` can be used to explicitly force data to be copied from the host to the device, or vice versa. Performing both computation and communication almost exclusively on the GPUs allows the code to avoid the overhead of data movement almost completely. The CPU host is effectively allowed to remain idle (except for some I/O), but with the official information [16] that 99% of each node's flops is on the GPUs, leaving the CPU idle does not lead to a significant "waste" of resources. On this machine, any attempt at overlapping operations on the CPU and GPU in the computations would lead to only very minimal (if any) benefit that would not justify the human effort or algorithmic complexity required to make it happen. As to be detailed later in Sec. 5, communications from the GPU are faster than those from the CPU, which is especially true when the requisite host-device data transfers are included in the cost of communicating from the CPU. The notion of performing FFT on some (already transposed) batches of data while performing communication on next batch was successful on Summit [6] because operations there could be profitably split and overlapped between the CPU and the GPU. However, overlapping is not appealing if there is a drastic difference in timing between the processes being overlapped, which is the case on Frontier.

Fig. 3 shows the essential lines of coding for a slabs code, focusing on global all-to-all communications within an OpenMP parallel region. Since all computations (including packing of data into contiguous buffers) are performed on the GPU, using host memory pointers with MPI would require a data copy from device to host before communication and a data copy from the host to device afterwards. In contrast, if MPI is performed with device memory pointers, these data exchanges between host and device are unnecessary. Regardless of which memory pointers are used, the arguments passed to the MPI_ALLTOALL are the same. In this example, since the intention is to swap $x-z$ planes to $x-y$ planes, the message size is the product of the complete data length in x and the sizes of partitions in y and z.

The largest problem size feasible on a given machine (or nodes thereof) is determined by the memory footprint of the code, which includes contributions from the number of variables stored and from memory allocated to intermediate arrays, such as communication buffers needed for MPI calls on non-contiguous data. The coding in Fig. 3 assumes the relevant physical variables are being transposed one at a time. If multiple variables are to be transposed together the buffers will be proportionately larger. In general, MPI communication involving a smaller number of larger messages is faster than that with a larger number of smaller messages, which would be more sensitive to latency versus bandwidth. However, the HPE Slingshot interconnect on Frontier offers very low latency, which allows us to benefit from communicating one variable at a time with reduced memory overhead, except at the most extreme problem sizes (to be discussed further in Sec. 5).

A pencils code follows similar principles but with additional considerations for row and column communicators. The forward transform sequence starts with data in pencils aligned in x, ready for a R-C transform in x. The data is then packed into a row buffer and transposed to pencils in z, via the row communicator that includes P_r MPI processes. A C-C transform is performed, followed by a transpose to pencils in y via the column communicator that includes P_c MPI processes. The pencils code requires two rounds of packing, all-to-all communications, and unpacking, but in the form of smaller bodies of data and over fewer MPI processes than in the slabs code.

Fig. 4 shows an example of OpenMP offloading implemented using the OpenMP `TARGET TEAMS DISTRIBUTE PARALLEL DO` construct which enables work-sharing via massive parallelism among multiple threads on the GPU. For code robustness, all variables referenced in the nested loop are declared as either `PRIVATE` to each thread which may hold different values simultaneously, or `SHARED` among the threads with values fixed throughout the loop. The OpenMP keyword `COLLAPSE` here effectively combines the four-level nested loop to provide greater opportunities for parallelization amongst the GCD's threads.

In the coding shown in Fig. 4 each GPU thread is able to update the elements of a destination array without conflict or duplication with work performed by other GPU threads sharing the memory of the arrays

```
!$OMP TARGET TEAMS DISTRIBUTE PARALLEL DO COLLAPSE(4) &
!$OMP       DEFAULT(NONE) PRIVATE(i,xp,yp,zp,y) &
!$OMP       SHARED(jproc,yjsz,isz, zjsz , jjst , iv , sndbuf_col , source)
do i=1,jproc
    do zp=1,zjsz
        do yp=1,yjsz
            do xp=1,xisz
                y=jjst(i-1)+yp-1
                sndbuf_col(xp,yp,zp,iv,i) = source(xp,y,zp,iv)
            end do
        end do
    end do
end do
!$OMP END TARGET TEAMS DISTRIBUTE PARALLEL DO
```

Fig. 4. Code fragment showing a nested loop performed on GPU using OpenMP constructs to pack pencils aligned in the y direction into a contiguous array that consists of "bricks" of data ready for an all-to-all exchange among MPI processes in the same column communicator. The variables $xisz$, $yjsz$ and $zjsz$ are equal to $(N/2)/P_r$, N/P_c and N/P_c respectively.

```
!$OMP TARGET DATA USE_DEVICE_PTR(vxz)
#ifdef  USE_NVIDIA
      call  cufftExecR2C(cu_plan_r2c ,C_LOC(vxz(1,1,1,ivar)),C_LOC(vxz(1,1,1,ivar)))
#elif  USE_AMD
      call   rocfft_execute  ( roc_plan_r2c ,  C_LOC(vxz(1,1,1,ivar)),C_NULL_PTR, C_NULL_PTR)
#endif
!$OMP END TARGET DATA
```

Fig. 5. Code fragment demonstrating the use of USE_DEVICE_PTR to instruct the compiler to use the GPU version of the vxz pointer when performing in-place real-to-complex FFTs on the GPUs.

```
!$OMP TARGET DATA USE_DEVICE_PTR(vxz)
      call  hipfftExecR2C( cu_plan_r2c ,C_LOC(vxz(1,1,1,ivar)),C_LOC(vxz(1,1,1,ivar)))
!$OMP END TARGET DATA
```

Fig. 6. Code fragment demonstrating the use of hipfftExecR2C instead of cufftExecR2C and rocfft_execute.

concerned. In contrast, in the case of data reduction operations (such as those that form partial sums on each MPI process), additional care is needed to prevent multiple GPU threads from accessing and updating the same elements of the destination array simultaneously (which would result in unpredictable outcomes). Currently, this situation requires use of the OpenMP `ATOMIC` construct, which ensures correct results by forcing each thread to wait for one another but leads to lower efficiency. However, by employing some additional intermediate buffers we can alleviate the extent of contention by reducing the number of threads simultaneously working on the same set of array memory positions. It is possible that future HPE CCE and AMD ROCm software releases will provide more elegant ways of performing OpenMP array reduction on the GPU safely.

2.3. Performing the 3D FFTs

Fig. 5 provides an example of a typical FFT execution call in the code. As in Fig. 3, the OpenMP directive `USE_DEVICE_PTR` instructs the compiler to use the pointer associated with the copy of the array in GPU memory when performing FFT computations. This code fragment includes compile-time options for both NVIDIA and AMD GPUs despite the fact that AMD's ROCm software includes a HIP layer that is designed to let the programmer use a single hipfft function call instead of explicitly including options for both cufft and rocfft in the code. The hipfft functions are simple pass-through functions (e.g. pass-through layer) that properly switch between cufft and rocfft within the ROCm library and allow developers to simplify their applications by using a single function call instead of multiple compile-time logical conditions to include the appropriate vendor-specific function call. The equivalent code fragment using hipfft is provided in Fig. 6. Early development of hipfft lagged slightly behind rocfft, and to make rapid progress during the initial extension of the codes to AMD GPUs, the rocfft functions were directly implemented into the GESTS family of codes. Future efforts on the GESTS codes may make the switch to hipfft now that it's robust and once the current research goals have been met.

Performance data on 3D FFTs are reported in Secs 4 and 5 as part of a performance analysis of the overall turbulence code, whose structure and requirements are first discussed in Sec. 3.

2.4. Requirements to build the code

We note here a final requirement for using the DNS codes on Frontier beyond the use of MPI, OpenMP Offloading, and an FFT package. Specifically, we need Fortran interfaces to access the C/C++ ROCm libraries. Instead of creating our own interfaces, we use the Fortran interfaces provided by AMD's ROCm HIPFORT package. Fortran is known to be non-ABI (Application Binary Interface) compliant, and to ensure ABI compliance between the GESTS codes and the Fortran interfaces in the compiled HIPFORT libraries, we build HIPFORT with the HPE/Cray Fortran compiler like we do for the GESTS codes.

3. The GESTS turbulence DNS code

3.1. Basic governing equations and numerical approach

While the governing Navier-Stokes equations for conservation of mass and momentum are well known (see. e.g. Ref. 38), for completeness we first summarize the main elements applicable to the instantaneous velocity field under the conditions of constant density and zero mean velocity, as:

$$\nabla \cdot \mathbf{u} = 0 \tag{4}$$

$$\partial \mathbf{u}/\partial t + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\nabla(p/\rho) + \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{5}$$

where p is the pressure fluctuation, ρ is the density, ν is the kinematic viscosity, and $\mathbf{f}$ is a forcing term [39,40] (which mostly acts at the large scales). The second term on the LHS represents convective effects, which are nonlinear in nature. Per Eq. (4), the velocity field is solenoidal or divergence-free, which in wavenumber space implies $\mathbf{u} \cdot \hat{\mathbf{k}} = 0$, i.e. the velocity vector in Fourier space be orthogonal to the wavenumber vector. With this constraint, and assuming $\mathbf{f}$ is also solenoidal, Eq. (5) can be written as

$$\partial \hat{\mathbf{u}}/\partial t = -\widehat{\nabla \cdot (\mathbf{u}\mathbf{u})}_{\perp \mathbf{k}} - \nu k^2 \hat{\mathbf{u}} \tag{6}$$

where the subscript $\perp \mathbf{k}$ denotes vectorial projection onto the plane perpendicular to $\mathbf{k}$. A Cartesian tensor form is

$$(\partial/\partial t + \nu k^2)\, \hat{u}_j = -ik_m P_{jl}(\mathbf{k})\, \widehat{u_l u_m} \tag{7}$$

which holds for $j = 1, 2, 3$, and $P_{jl}(\mathbf{k}) = \delta_{jl} - k_j k_l / k^2$ with δ_{jl} being the Kronecker delta tensor. We use the well-known pseudo-spectral algorithm of Rogallo [41], which uses a modified pressure that allows the number of independent nonlinear terms that need to be stored to reduce from 6 (in $\{u_l u_m\}$) to 5. Integration in time is implemented using an exact integrating factor for linear terms on the LHS, and second-order Runge-Kutta (predictor-corrector) for nonlinear terms on the RHS. Aliasing errors for the nonlinear terms are controlled by a combination of phase shifting and truncation at wavenumbers beyond $k_{max} = \sqrt{2}N/3$ in Fourier space [18]. Numerical stability requires the time step Δt be chosen such that the Courant number

$$C = \Delta t \left[\frac{|u|}{\Delta x} + \frac{|v|}{\Delta y} + \frac{|w|}{\Delta z} \right]_{max} \tag{8}$$

(where the maximum is taken over all grid points) is always below 1.0.

3.2. GPU implementation, memory and I/O

As noted earlier, in our code on Frontier the GPU performs almost all of the work, with an OpenMP offload programming model, except mainly for I/O operations including writing checkpoints and reading data upon restarting. Each Runge-Kutta substep starts with velocity in Fourier space. A phase shift is applied as part of aliasing error control, and all three velocity components are then transformed to physical space (in the order y, z, x as described in Sec. 2) where nonlinear terms are formed. The nonlinear products are then transformed to wavenumber space, where differentiation operators and an inverse phase shift are applied to obtain the desired convective terms on the RHS of Eq. (7). The numerical solution is then advanced in time according to the Runge-Kutta scheme chosen. At the predictor step we also form (when the velocity field in the physical space) the maximum value of the square bracket in Eq. (8), which is used to calculate Δt conforming to a user-specified value of the Courant number. Integrating factors and numerical forcing are also applied in the predictor step only. When the predictor-corrector cycle is complete some statistics are computed if desired. Then the same process is repeated for the next time step.

A low memory footprint is essential to the feasibility of larger problem sizes on any given machine. The main memory requirements in our DNS code include the memory to store Fourier coefficients of the solution variables (i.e. the velocity components), the nonlinear terms and any partial updates depending on the chosen Runge-Kutta scheme in time, as well as contiguous sending and receiving communication buffers based on the number of variables (say, Q) processed in each MPI all-to-all communication call. In Rogallo's formulation, the second-order Runge-Kutta integration in time requires only 8 memory positions at each wavenumber mode, with 3 and 5 for the velocity vector and the nonlinear terms, respectively. The MPI all-to-all library call requires contiguous buffers for both sending and receiving data of size $2Q$ variables per grid point or wavenumber mode. In the case of the pencils decomposition, this requirement applies to both the row and column communicator. However, since these row and column buffers are not used at the same time, memory buffers for just the row communicator can be allocated on the GPU and reused for the column communicator. The shape of the send and receive buffers can be altered dynamically using the Fortran intrinsic function, C_F_POINTER. Finally, a single real-valued variable is used at each wavenumber mode to enforce truncation beyond k_{max}.

Since the number of Fourier modes represented in the code is $N^3/2$, at 8 bytes per single-precision complex number the total memory required per MPI process (hence each GCD on Frontier) is very close to

$$(8 + 2Q + 0.5) \times (N^3/2P) \times 8/(1024)^3 \tag{9}$$

if measured in units of 1 GiB. This is indeed verified (to within a small fraction of 1 GiB) by our actual measurements. In particular, the choices $Q = 1$, $N = 2048$ and $P = 8$ give 42 GiB per GCD. With N almost always a pure power of 2 except for a possible factor of 3 or 9, this estimate also establishes 2048^3 as the largest problem size that can fit into a single node on Frontier, and that a 32768^3 problem will require at least 4096 nodes.

It is worth noting that, at the end of each time step, after the velocity has been updated to t_{n+1}, the memory positions used for nonlinear terms are freed up and can be exploited for additional calculations not required for the time advance itself. For instance, for detailed studies of fine-scale intermittency in turbulence, it is useful to obtain the fluctuating energy dissipation rate and enstrophy, defined respectively as:

$$\epsilon = 2\nu s_{ij} s_{ij} \quad ; \quad \Omega = \omega_i \omega_i \tag{10}$$

where $s_{ij} = (\partial u_i/\partial x_j + \partial u_j/\partial x_i)/2$ is the strain rate tensor and ω_i is the ith component of the vorticity vector $\omega = \nabla \times \mathbf{u}$. While the definitions of ϵ and Ω both involve summation over multiple terms, the summations can be applied incrementally without new memory requirements. A small increase in memory, at 4 GiB per GCD, is needed for calculating pressure fluctuations by solving the Poisson equation

$$\nabla^2(p/\rho) = -\frac{\partial^2 u_i u_j}{\partial x_i \partial x_j}\,, \tag{11}$$

which can be transformed to Fourier space as $-k^2 \hat{p}/\rho = k_i k_j \widehat{u_i u_j}$ and solved in a pseudo-spectral manner with aliasing error removal. Computing the statistics of ϵ, Ω and p directly (as on-the-fly processing) in the DNS code gives more information on a time-resolved basis [42] in contrast to post-processing of a limited number of velocity fields saved at time intervals many small-eddy time-scales apart.

Since our simulations are of a time-evolving nature that often requires many tens of thousands of time steps, a flexible and space-friendly approach for checkpointing and restarting is important. To save on disk space, we write checkpoints in Fourier space, where Eq. (4) becomes $k_1 \hat{u}_1 + k_2 \hat{u}_2 + k_3 \hat{u}_3 = 0$. This allows us to write mainly just two velocity components and to recover the third if the corresponding wavenumber is nonzero, as in

$$\hat{u}_2 = -(k_1 \hat{u}_1 + k_3 \hat{u}_3)/k_2 \quad (\text{if} \quad k_2 \neq 0)\,. \tag{12}$$

The total volume of data written on the disk per checkpoint is close to $8N^3$ bytes, which is 256 TiB (or 1/4 PiB) if $N = 32768$. To avoid causing adverse effects that such a large I/O data volume can have on the performance of a large filesystem shared with other users, one approach is to collect data from P parallel processes into a smaller number (say M) of files, each associated with P/M processes and placed into a set of sub-directories. However, collecting data from multiple parallel processes in the code requires an MPI_GATHER communication call, which will need an extra buffer and hence use more memory. Instead, on Frontier we use MPI-IO, in which each parallel process writes data

Table 2
Elapsed wall time per step (in seconds) observed for the *slabs* version of the full turbulence DNS code, at minimum node counts meeting memory requirements at each problem size, with a breakdown into costs for several key operations in each case. 8 MPI ranks and 8 GPUs are used on each node. P+U denotes packing and unpacking related to all-to-all communication. Weak scaling is calculated with respect to 4096^3 on 8 nodes (denoted by WS1) or with respect to each smaller problem size with N halved (WS2). The first 5 data points in this table were obtained on a relatively quiet system in April 2023. The last was from a more recent test in April 2024. A change in library versions on the system may have contributed to the differences between numbers in the last two rows.

N^3	#Nodes	#MPI	FFT	P+U	MPI	Other	Total	WS1(%)	WS2(%)
2048^3	1	8	1.360	0.243	1.846	0.643	4.09	–	–
4096^3	8	64	1.409	0.246	7.762	0.685	10.11	–	–
8192^3	64	512	1.580	0.255	8.152	0.764	10.75	101.8	101.8
16384^3	512	4096	2.196	0.268	8.360	0.700	11.53	102.2	100.4
32768^3	4096	32768	3.423	0.276	8.621	0.747	13.07	96.6	94.5
32768^3	4096	32768	2.708	0.246	8.482	0.633	12.07	104.6	98.9

segments of specified size starting at a prescribed location (known as the "displacement") in the file. Since MPI-IO does incur some cost for communication (with details hidden from the user), the best performance is expected if the underlying communication is wholly within the node. This is achieved by choosing P/M to be equal to the number of MPI processes (8 for Frontier) on each node, giving 1 file per node. On Frontier, MPI-IO can be executed either on the CPU or the GPU. Finally, at extreme problem sizes, the load imposed on the filesystem by thousands of files being open and simultaneously written by even more MPI processes can also cause an I/O traffic jam that impairs performance. This scenario can be avoided by a relay scheme that say, divides P MPI processes into two groups of size $P/2$, and forcing the second group to wait for a signal from the first before they are allowed to proceed.

As mentioned earlier, we have both slab and pencil implementations, and for flexibility, it is highly desirable that these codes can take the same input and write the same output, regardless of the shape of the actual $P_r \times P_c$ process grid geometry when pencils are used. This goal is achieved by having both codes write checkpoints in the form of either one file per slab (for the MPI_GATHER method) or one file per node (for the MPI-IO method). This approach also simplifies the coding necessary for reading checkpoints, especially if using MPI-IO.

4. Performance data and analysis

In this section, we report on the performance of the code, including timings per step, estimates of GPU speedup relative to a CPU-only code of similar structure, and both weak scaling and strong scaling over a range of problem sizes and node counts. Since we were granted access to the machine earlier than others through the CAAR Program for Frontier, we have collected, since February 2023, benchmarking data associated with several releases of the CCE compilers, ROCm libraries, and MPI libraries. All of the timings reported below correspond to the CCE compiler 15.0.1, ROCm 5.4.3, and cray-mpich 8.1.23. For OpenMP we use features at version 4.5 or higher. All MPI AlltoAll communications are performed on the GPU unless stated explicitly otherwise.

4.1. Data collection and assessment of scalability

For a critical evaluation of large user-production codes, it is important that a set of rational measures of performance be adopted. We describe and explain our practices as below.

The simplest measure of code performance in our application is the elapsed wall time obtained readily from MPI_WTIME library calls at the start and end of each time step. Although performance monitoring tools available on most major systems can provide a variety of lower-level performance data, strategic placements of MPI_WTIME calls in the user code itself are very helpful in separating the costs of specific categories of operations in the code. We only use performance data from regular time steps free of additional operations such as initialization, on-the-fly statistical processing, and checkpointing, which all tend to be time-consuming but are performed only at a small fraction of the total number of time steps in long production simulations especially at high grid resolutions. In principle, we take the longest timings over all MPI ranks, but since our codes have no significant load imbalance, all MPI processes report nearly the same timings for regular time steps.

Although variability among the MPI processes is minimal, communication traffic due to network contention with other jobs on the system can still cause variations between different time steps. These runtime variations are usually small in production conditions but may become substantial in configurations dominated by communications involving numerous small messages at large node counts. When this variability is minimal, it is appropriate to report performance data using the average of elapsed time over a certain number of time steps, with the expectation that the average will converge with increasing number of steps. However, when variability of an erratic nature arises due to network contention, MPI communication timings may also behave like a random variable characterized by substantial fluctuations above a lower bound. The fluctuations can unpredictably alter the observed average runtime over a finite number of steps, while the lower bound in communication timings is directly associated with the true (if somewhat theoretical) measure of performance capability attainable by the code. Most of the data points in Table 2 were obtained in a restricted early-user period when contention from other jobs was minimal. For these cases, the timings represent the average of the runtimes over 10 successive iterations. In contrast, recent timings in other tables shown in this section were obtained with Frontier subjected to very high loads and greater network variability. Consequently, in these latter tables we have used timings from the best out of roughly 10 time steps, which gives a more accurate measure of the performance attainable by the present codes as discussed above. We also report results from internal execution profiling in the code that allow the wall time for these best time steps to be decomposed into contributions from major operations, such as 1-D FFT library calls, all-to-all communication, packing and unpacking operations, and all other computations (including forming nonlinear products and advancing solution variables in time).

From a scalability perspective, although the concepts of weak scaling towards larger problem sizes and strong scaling towards shorter runtimes are well known, different choices of reference problem size or node count can give different results. In particular, it is possible to assess weak scaling relative to the smallest problem size tested or considered appropriate for the hardware. For Frontier, the largest problem size that can be simulated on one node is 2048^3. However, because of a great contrast between communication speeds within the node or across different nodes (absent in a single node problem), a more appropriate reference point is actually 4096^3 on 8 nodes. At the same time, since (perhaps unlike finite difference methods) changes in problem sizes are

Table 3
Elapsed wall time per step (in seconds) observed for the *pencils* version of the DNS code, presented in a manner similar to that for the slabs version in Table 2, but including data for several shapes ($P_r \times P_c$) of the 2D processor grid employed. Timings are taken from the best time step in each case. Time spent on MPI_ALLTOALL is split into contributions from row and column communicators. Weak scaling is assessed in the same manner as in Table 2, relative to case IDs enclosed in square brackets, with the row communicator size held fixed.

	N^3	#Nodes	P_r	P_c	FFT	P+U	R-MPI	C-MPI	Other	Total	WS1(%)	WS2(%)
1	2048^3	1	2	4	1.81	1.08	0.74	1.27	0.58	5.48		
2	2048^3	1	1	8	1.65	0.66	0.00	1.83	0.42	4.55		
3	4096^3	8	8	8	2.03	1.09	2.08	5.97	0.82	11.98		
4	4096^3	8	4	16	1.77	0.89	1.52	5.95	0.52	10.64		
5	4096^3	8	2	32	1.89	0.74	0.74	6.86	0.45	10.68		
6	4096^3	8	1	64	1.78	0.43	0.00	7.00	0.43	9.62		
7	8192^3	64	8	64	1.55	0.76	2.09	8.15	0.48	13.03	99.6 [3]	99.6 [3]
8	8192^3	64	4	128	1.50	0.65	1.52	8.18	0.44	12.28	93.7[4]	93.7 [4]
9	8192^3	64	2	256	1.46	0.60	0.74	8.15	0.43	11.39	101.6 [5]	101.6 [5]
10	8192^3	64	1	512	1.56	0.40	0.00	8.14	0.43	10.52	99.1 [6]	99.1 [6]
11	16384^3	512	8	512	2.09	0.66	2.08	9.26	0.53	14.61	95.7 [3]	96.1 [7]
12	16384^3	512	4	1024	2.19	0.61	1.51	9.04	0.49	13.85	89.5 [4]	95.5 [8]
13	16384^3	512	2	2048	2.34	0.59	0.74	8.77	0.48	12.92	96.5 [5]	95.0 [9]
14	16384^3	512	1	4096	2.57	0.40	0.00	8.32	0.43	11.71	95.8 [6]	96.7 [10]
15	32768^3	4096	8	4096	3.09	0.67	2.08	9.36	1.03	16.21	92.1 [3]	96.2 [11]
16	32768^3	4096	4	8192	3.32	0.61	1.52	8.91	0.54	14.89	89.2 [4]	99.7 [12]
17	32768^3	4096	2	16384	3.41	0.60	0.74	9.72	0.49	14.95	88.9 [5]	92.1 [13]
18	32768^3	4096	1	32768	3.76	0.40	0.00	8.50	0.46	13.10	91.7 [6]	95.8 [14]

more discrete than continuous, it is also useful to measure weak scaling by using the next smaller case (say a fixed factor of 2 smaller in N), instead of the smallest case. A similar consideration applies to strong scaling, relative either to the smallest node count that can accommodate a specific problem size, or to the closest smaller node count (i.e. P versus $P/2$). For both weak scaling and strong scaling, we provide numbers from the two definitions as described here.

4.2. Slabs code and absolute performance

Table 2 shows overall performance and weak scaling for the slabs code, which is slightly faster than the pencils version. Weak scaling is calculated by assuming the computational workload is, through the use of 1-D FFT calls, proportional to $N^3 \log_2 N$ (based on N^2 lines of length N each) for a given N^3 problem. It is clear that MPI communication dominates the runtime, and that its scalability is low between 2048^3 on 1 node to 4096^3 on 8 nodes, but very good from 8 nodes onwards. This is explained by communication within the code being much faster than between different nodes on Frontier. Beyond 8 nodes, the communication cost becomes dominated by inter-node traffic, which, although expensive, does scale quite well up to large node counts. The next most costly operation is 1-D FFT performed using the AMD-provided ROCm library. The cost of 1-D FFTs depends somewhat on whether the transforms are performed on strided versus non-strided data, using in-place or out-of-place storage, and certain properties of the ROCm library itself. The computational cost of 1D FFTs per process scales as $(N^3/P)\log_2 N$. Under weak scaling conditions, where N^3/P remains constant, the cost would scale as $\log_2 N$. The table shows that this scalability holds for problem sizes up to 8192^3. We do observe some degradation at problem sizes 16384^3 or higher, which is related to an existing artifact of the AMD ROCm library affecting large transform lengths. Timings incurred by packing, unpacking, and various other operations performed by user-written OpenMP offloading are seen to scale very well, while usually accounting for only 5% or less of the total time per step.

Besides scalability on a given machine, an important consideration for science is to what extent a new algorithm can enable larger simulations to be performed in comparable or less time on a new machine, versus an older but still powerful system with some similarities at least in the type of hardware. For us, a comparison of slabs codes on Summit (where slabs are favored) and Frontier is most appropriate since both are leadership-class GPU machines albeit with different hardware and from different, but sequential, technological generations. For a given elapsed wall-clock time $\mathcal{T}$, a convenient figure of merit, here denoted by χ, is proportional to the volume of work performed per unit time, $N^3/\mathcal{T}$, where the factor of $\log_2 N$ in typical FFT cost estimates is omitted in view of the dominance of communication costs, which are nominally proportional to the number of grid points. If weak scaling is 100% then $\chi \propto N^3$ with $P \propto N^3$. On Frontier, 12.07 secs for $N = 32768$ using 4096 nodes in Table 2 above gives $\chi \approx 2.91 \times 10^{12}$, which is 6.6x higher than 4.40×10^{11} obtained from 14.24 secs for $N = 18432$ using 3072 Summit nodes in [6] (Table 3 therein). This factor of 6.6x also exceeds the 5.9x differential in χ between our largest problem sizes on Frontier (32768^3) and on Summit (18432^3) that is reported in [16] (Table 6 therein).

4.3. Weak scaling: pencils code and GPU speedup

Although the slabs code may be slightly faster in some cases, the pencils code has the advantage of greater flexibility when an increase in node count is necessary, either to accommodate additional science-based memory requirements, or to achieve faster solution times for a given problem. The pencils code is also the better choice for inferring CPU-to-GPU speedup, since CPU codes running at high resolution often require a very large number of MPI processes with $P > N$ and a pencils decomposition.

Table 3 shows a detailed compilation of pencils code performance data over a range of problem sizes for N varied in powers of 2 from 2048 to 32768, and for several choices of the $P_r \times P_c$ row-and-column processor grid geometry at the minimum node count needed to satisfy the memory requirements (per Eq. (9)) in each case. For the code to run with 1 MPI process per GCD, and to avoid splitting row communicators between multiple nodes, we limit P_r to a maximum of 8, while P_c can be up to N. Similarly to the slabs code, the case of 2048^3 is extremely efficient since it does not involve inter-node communication traffic, which also causes weak scaling from 2048^3 to 4096^3 to be artificially low. Incidentally, the fact that such a single-node data point is entering the discussion is a reflection of the power of individual nodes on modern GPU machines such as Frontier.

It is clear that the shape of the processor grid geometry has significant impact on performance. In particular, for each N and P combination in the table, the cost of row-communicator MPI decreases system-

Table 4
Selected timings (in seconds) obtained from a pencils CPU code with the same overall structure as the GPU version.

Case	N^3	#Nodes	P_r	P_c	FFT	P+U	MPI	Other	Total
1	2048^3	1	2	32	23.1	37.5	16.3	27.4	104.3
2	2048^3	1	8	8	19.5	33.5	14.8	28.1	95.9
3	2048^3	1	64	1	19.5	36.1	18.0	29.5	103.1
4	4096^3	8	64	8	19.3	35.0	21.8	30.9	107.0
5	4096^3	8	8	64	22.1	36.9	21.0	30.9	109.7
6	8192^3	8	64	64	20.9	33.3	22.6	28.1	104.9

atically as P_r is reduced through the sequence $\{8, 4, 2, 1\}$, by about 27% from $P_r = 8$ to $P_r = 4$, and about 51% from $P_r = 4$ to $P_r = 2$. This reduction is a consequence of the design of the network hardware within each Frontier node, that a slabs code cannot benefit from. In the case of $P_r = 1$ further savings are possible since the row-communicator MPI can be avoided altogether (with zero time taken) and the packing and unpacking necessary for ensuring contiguous messages are likewise rendered superfluous. Some differences between the pencils code with $P_r = 1$ and the slabs code do remain: in the use of in-place versus out-of-place transforms for real-to-complex or complex-to-real FFTs in the x direction, and in the cost of packing and unpacking related to the use of strided FFTs. Nevertheless, a comparison between Tables 2 and 3 shows that the best overall performance obtained from the pencils code with ($P_r = 1$) at 4096^3 and 8192^3 is slightly faster than for the slabs code. It is also encouraging that at 16384^3 the pencils code timing with $P_r = 1$ differs from that of the slabs code by less than 2%, which is within the margins of normal variability on the machine, while the total communication times are essentially the same. Weak scaling calculated on the basis of computational workload, i.e. in proportion to $N^3 \log_2 N$ is seen to be very high, being close to 90% for 32768^3 on 4096 nodes relative to 4096^3 on 8 nodes, and exceeding 95% for each 2x increase in N beyond 4096 in the table.

The cost of column-communicator MPI_ALLTOALL has a more complicated behavior in contrast to the row-communicator, presumably because P_c often spans a tremendous range as the problem size increases. This cost is nearly independent of P_c at 8192^3 but there is evidence of variability at 16384^3 and 32768^3. Since this communication is the most time-consuming operation, it is useful to consider the peer-to-peer (P2P) message size involved, the number of communication calls per time step, and the nature of variability potentially associated with network contention on the system. For the column communicator, each MPI process holding real-valued flow variables at N^3/P grid points (or half as many complex-valued Fourier coefficients) has to send and receive messages, per variable, of size $4N^3/PP_c$ in bytes assuming single precision. In weak scaling comparisons, the factor N^3/P is held constant while $P_c \propto N^3/P_r$. As a result, the message size becomes small especially when a small value of P_r (hence large P_c) is used at the largest problem sizes: e.g. the message size per variable is 0.125 MiB for Case 18, compared to 0.5 MiB for Case 16. The structure of our pseudo-spectral algorithm is such that at each of two Runge Kutta substeps in a single time step the number of flow variables passed through the column communicator are 3 and 4 for inverse and forward transforms respectively. The total number of column communicator all-to-all calls per time step (if handling one variable per call) is thus 14. We are able to check that, in cases that showed very good weak scaling, the total column communicator time per step is indeed very close to 14 times the cost of a single call, with minimal deviations. However, in some tests at large node counts some communication calls have been observed to be up to several times slower, in a manner which is random, and consistent with variability typical of network contention.

It is worth noting that, a comparison of Tables 2 and 3 shows that at 32768^3 using 4096 nodes, the communication cost in the pencils code with $P_r = 1$ matches that in the slabs code quite closely, but the pencils code takes up to 1 second longer in most cases because of a difference in 1-D FFT performance. A closer inspection shows, in fact, that the FFTs in the pencils code with $P_r = 1$ are slower than those in the slab code for all but one of the five problem sizes; the exception being for 8192^3 on 64 nodes. While the vector strides used in the C-C transforms are a potential factor, we first note that FFT library calls may, in general, be classified as in-place (results stored in the same array as the input data) or out-of-place (results stored in a new buffer which requires additional memory). In our slab code (similar to that in [6]), we store Fourier coefficients for an N^3 problem in arrays of shape $(N/2+1, N, N)$, with all transforms taken in-place, and vector strides being 1 for C-R and R-C transforms in x, but $N/2+1$ for C-C transforms in y and z. However, the pencils code is built upon a more complex structure with Fourier coefficients in arrays of different shapes depending on whether the data (e.g. pencils) are aligned in x, y or z directions at different points in the code. These complexities are most readily accommodated by out-of-place R-C and C-R transforms with stride 1 in x, and in-place C-C transforms with stride $N/(2P_r)$ in y and z. We have found that R-C and C-R transforms in the slab and pencil codes give essentially the same performance (presumably because strides are 1 in both cases). It can be seen that at $N = 16384$ and higher, FFTs become slower as P_r decreases and vector stride increases. At the same time, at the limit of $P_r = 1$, C-C transforms in the pencil code are significantly slower than those in the slab code although the pencils code uses a vector stride that is smaller than the one used in the slab code ($N/2$ vs $N/2+1$). In our best understanding, this is the result of some intricacies of rocFFT library versions presently available on Frontier. It is possible that the rocFFT software is currently optimizing some transform sizes and configurations differently than others. Since rocFFT is still under active development, we expect future ROCm releases to resolve this behavior.

Another measure of GPU code performance is the speedup obtained from using the GPU as an accelerator to the CPU. CPU codes can, of course, be written in pure MPI mode where every core available on the CPU host operates as a distinct MPI process, or in a hybrid mode where each core operates as an OpenMP thread, often with only so-called master threads performing MPI calls while all threads share in the computational workload. For inference of the CPU to GPU speedup, we choose pure MPI for CPUs since a direct comparison is needed only at modest problem sizes, where any benefit from OpenMP multi-threading is likely to be limited. Table 4 shows the CPU performance data, where the standard FFTW library is used for FFTs. Since each node on Frontier consists of 64 cores, the overall processor count $P = P_r \times P_c$ shown is a multiple of 64 in each case. It appears that $P_r = 8$ gives the best MPI performance. Comparisons with the best GPU timings at 2048^3, 4096^3 and 8192^3 (Cases 2, 6, 10 in Table 3) give a GPU speedup factor on the order of 20 at 2048^3 on a single node and on the order of 10 at larger problem sizes run on multiple nodes. At 8192^3 resolution (Case 10 of Table 3 to Case 6 of Table 4) the GPU speedup factors for FFT, Pack+Unpack, (total) MPI, and Other contributions to the cost are roughly 13x, 33x, 3x, and 65x respectively. Since the Pack+Unpack and Other contributions are essentially all low-intensity compute-only operations, these large speedup factors show that the user-written OpenMP offloading in our GPU coding is highly efficient.

Table 5
Assessment of strong scaling for the pencils code, at grid resolution 8192^3 and node counts up to 8 times of that required by memory, with P_r varied from 8 to 1. Percentage strong scaling is calculated by referring to the smallest node count for each case (shown as SS1) or node count smaller by a factor of 2 (shown as SS2).

Case	N^3	#Nodes	P_r	P_c	FFT	P+U	R-MPI	C-MPI	Other	Total	SS1(%)	SS2(%)
1	8192^3	64	8	64	1.55	0.76	2.09	8.15	0.48	13.03	–	–
2	8192^3	128	8	128	0.79	0.34	1.05	4.50	0.27	6.96	93.6	93.6
3	8192^3	256	8	256	0.40	0.17	0.53	2.24	0.13	3.46	94.1	100.6
4	8192^3	512	8	512	0.20	0.08	0.27	1.02	0.06	1.63	99.9	106.1
5	8192^3	64	4	128	1.50	0.65	1.52	8.18	0.44	12.28	–	–
6	8192^3	128	4	256	0.76	0.31	0.76	4.52	0.23	6.59	93.2	93.2
7	8192^3	256	4	512	0.38	0.16	0.38	2.15	0.12	3.18	96.5	103.6
8	8192^3	512	4	1024	0.19	0.08	0.19	1.01	0.06	1.53	100.3	103.9
9	8192^3	64	2	256	1.46	0.60	0.74	8.15	0.43	11.39	–	–
10	8192^3	128	2	512	0.73	0.30	0.37	4.46	0.23	6.09	93.5	93.5
11	8192^3	256	2	1024	0.37	0.15	0.19	2.12	0.11	2.94	96.9	103.6
12	8192^3	512	2	2048	0.18	0.08	0.09	1.13	0.06	1.55	91.9	94.8
13	8192^3	64	1	512	1.56	0.40	0.00	8.13	0.42	10.52	–	–
14	8192^3	128	1	1024	0.78	0.20	0.00	4.27	0.21	5.47	96.2	96.2
15	8192^3	256	1	2048	0.39	0.11	0.00	2.03	0.11	2.64	99.6	103.6
16	8192^3	512	1	4096	0.20	0.05	0.00	0.93	0.05	1.24	106.1	106.5

Table 6
Similar to Table 5, but for resolution 16384^3. Cases marked by asterisks were run with the parameter $Q = 4$ as discussed in the text.

Case	N^3	#Nodes	P_r	P_c	FFT	P+U	R-MPI	C-MPI	Other	Total	SS1(%)	SS2(%)
1	16384^3	512	8	512	2.09	0.66	2.08	9.26	0.53	14.61	–	
2	16384^3	1024	8	1024	1.05	0.33	1.05	4.36	0.26	7.05	103.6	103.6
*3	16384^3	2048	8	2048	0.53	0.17	0.53	2.27	0.13	3.62	100.9	97.4
4	16384^3	512	4	1024	2.19	0.61	1.51	9.04	0.49	13.85	–	–
5	16384^3	1024	4	2048	1.19	0.30	0.76	4.16	0.24	6.57	105.4	105.4
*6	16384^3	2048	4	4096	0.55	0.15	0.38	2.21	0.12	3.42	101.3	96.1
7	16384^3	512	2	2048	2.34	0.59	0.74	8.77	0.48	12.92	–	–
8	16384^3	1024	2	4096	1.18	0.29	0.37	4.27	0.24	6.36	101.6	101.6
*9	16384^3	2048	2	8192	0.59	0.15	0.19	2.26	0.14	3.32	97.3	95.8
10	16384^3	512	1	4096	2.57	0.40	0.00	8.32	0.43	11.71	–	–
11	16384^3	1024	1	8192	1.28	0.22	0.00	4.06	0.20	5.76	101.6	101.6
*12	16384^3	2048	1	16384	0.65	0.10	0.00	2.02	0.12	2.90	100.9	99.3

4.4. Strong scaling: pencils code

To investigate strong scaling for the pencils code with allowance for the effects of the processor grid geometry, we performed tests with P_r fixed but P_c and P increased by factors of 2, 4, and 8. We focus our attention on the three largest problem sizes, namely 8192^3, 16384^3 and 32768^3, subject to the constraints that the largest allowable values of P_c and P in our work are equal to N and 8192 respectively. In the strong scaling study, as P_r is held constant and P_c is doubled for a given problem size, the problem is being distributed across twice the number of MPI ranks (recall $P = P_r \times P_c$) than the preceding case. This distribution leads to half the number of grid points per MPI rank and half the volume of data per MPI rank to exchange during the MPI AlltoAll communications. Likewise, the communication times are ideally expected to decrease by half as the number of nodes are doubled for a given problem size.

Table 5 shows strong scaling data for 8192^3 grid points with node counts 64 to 512. Clearly, the strong scaling achieved is excellent, being higher than 93% upon each doubling of node count for all four possible values of the row-communicator dimension tested. The scalability is uniformly good in all types of operations, including 1-D FFT library calls, pack and unpack, all-to-all in the row and column communicators, and other user-written computational loops. Three out of four values of P_r (except 2) actually show 100% strong scaling or better for an 8x increase in node count. The fact that this excellent scalability is achieved even as communication costs account for some 75% of the execution time indicates HPE Slingshot indeed performs very well at least at the scale of the problem size referenced in this table (although this does not rule out limitations potentially arising at yet-larger problem sizes).

Table 6 shows results at 16384^3. The range of node counts accessible is limited to a factor 4, as the result of the requirement $P_c \leq N$. High scalability similar to that for 8192^3 in Table 5 is readily obtained for a 2x increase of node count from 512 to 1024. However, a further increase from 1024 to 2048 nodes for this problem size did pose some challenges, in the form of substantial variability in timings between successive time steps, which is (perhaps unsurprisingly) associated with communication across a large number of nodes. The variability is more significant at smaller values of P_r, which in turn requires large P_c. For a given N, large P_c also implies smaller message sizes, potentially falling outside the bandwidth-limited regime of MPI performance (to be analyzed further in Sec. 5).

To mitigate the MPI-related issue above, and to increase the likelihood of sustained stable performance, we have implemented a feature in the code to allow a smaller number of calls to the column-communicator MPI_ALLTOALL with larger message sizes. Specifically, instead of the standard choice $Q = 1$ in Eq. (9) we let Q be raised in connection to the number of variables that must be transposed at specific locations in the code. For simulations of the velocity field only, the number of flow variables that require transpose via the column communicator is 3 as part of a 3D FFT from wavenumber space to physical space, and 4 from physical space to wavenumber space. Use of a larger Q is also subject to memory constraints since the memory requirement will increase, in

Table 7
Strong scaling at 32768^3 grid resolution, shown in a format similar to Table 6, but for 32768^3 grid solution and including a direct comparison between performance, obtained using $Q = 1$ versus $Q = 4$. At this resolution only 2 node counts (4096 and 8192) are feasible. The strong scaling percentage is reported for data obtained using $Q = 4$ versus the case number shown indicated in square brackets. All 8192-nodes timings in this table were obtained from a single batch job using the same nodes for all cases listed.

Case	N^3	Nodes	P_r	P_c	Q	FFT	P+U	R-MPI	C-MPI	Other	Total	SS(%)
1	32768^3	4096	8	4096	1	3.09	0.67	2.08	9.36	1.03	16.21	
2	32768^3	8192	8	8192	1	1.55	0.33	1.05	7.33	0.29	10.55	
3	32768^3	8192	8	8192	4	1.55	0.33	1.05	7.02	0.24	10.19	79.6 [1]
4	32768^3	4096	4	8192	1	3.32	0.61	1.52	8.91	0.54	14.89	
5	32768^3	8192	4	16384	1	1.67	0.30	0.76	7.21	0.25	10.20	
6	32768^3	8192	4	16384	4	1.67	0.30	0.76	7.34	0.29	10.36	71.8 [4]
7	32768^3	4096	2	16384	1	3.41	0.60	0.74	9.72	0.49	14.95	
8	32768^3	8192	2	32768	1	1.73	0.30	0.37	7.32	0.27	9.98	
9	32768^3	8192	2	32768	4	1.73	0.30	0.37	7.31	0.28	9.98	74.9 [7]

the order of $8(Q-1)(N^3/P)/(1024^3)$ in GiB per GCD. This means use of $Q > 1$ is both easier and more beneficial in runs that emphasize strong scaling at larger problem sizes. The timings from runs with 2048 nodes in Table 6 were obtained with $Q = 4$. It can be seen that the scalability obtained in this manner is nearly ideal. However, variability in occasional slow time steps is still present, especially at small P_r and large P_c. Although the 1×16384 case (# 12 in this table) is timed to be the fastest, the best choice for production runs with stable performance is actually 4×4096 (Case #6).

Table 7 shows strong scaling data for the most demanding test i.e. 32768^3 from 4096 to 8192 nodes. At this scale, since P_c cannot exceed 32768, P_r can only be 2, 4, or 8. The costs of FFT, packing/unpacking, and row-communicator MPI are independent of Q, and are scaling ideally, taking half as long when the node count is doubled. The strong scaling overall is in the range 70–80%, lower than that seen in Table 6 mainly because of imperfect strong scaling of the column communicator MPI. This reduction of scalability occurring near the extreme limits of machine capability is not unexpected, since a large increase in the MPI process count causes both reduced message sizes and greater susceptibility to network contention on the system. At this problem size, it seems that benefit from use of $Q = 4$ is only modest, at least if data from all possible values of P_r are considered. However, the increase in message sizes (with a matching reduction in the number of MPI calls) does lead to greater performance stability: although not shown in the table, the elapsed wall-time variation among 8 time steps is up to 3 to 5 secs per step for cases with $Q = 1$, but is reduced to less than 0.1 sec. by using $Q = 4$. The effect of message sizes is further investigated in Sec. 5.

For an overall assessment of the performance of our pencils production code, we show in Fig. 7 the best timings obtained for each problem size (N^3) and node count ($P/8$) combination that has been tested in detail in this work. Essentially, for each problem size, we have varied the node count over a factor of 8 provided P_c does not exceed N and the number of nodes used does not exceed the number of nodes available on the system. Timings are taken from the best processor grid $Pr \times P_c$ available, while also considering the effect of the maximum number of variables exchanged via an MPI_ALLTOALL over P_c parallel processes forming the column communicator in our pencils DNS code. Strong scaling can be assessed readily with reference to the smallest node count necessary to provide sufficient memory, except for the case of 2048^3 where all work was performed on a single node with extremely fast communication. It is remarkable that, unlike common experiences in large-scale computing, as problem size increases from 2048^3 to 16384^3, the strong scaling not only does not degrade but actually improves, although at our extreme-scale problem size of 32768^3 strong scaling does fall off appreciably between 4096 and 8192 nodes. Weak scaling between 16384^3 and 32768^3 is also impacted by two secondary factors: namely (i) effects of striding on FFT performance at large problem sizes, (ii) a tendency for an optimal P_r shifting towards larger P_r (say from 1 to 2, 4 or 8 instead) at problem sizes larger than 8192^3. The first is a software artifact which will likely see improvement from a future version of the AMD ROCm library. The second is a consequence of cross-node message sizes becoming small in the column communicator as P_r decreases.

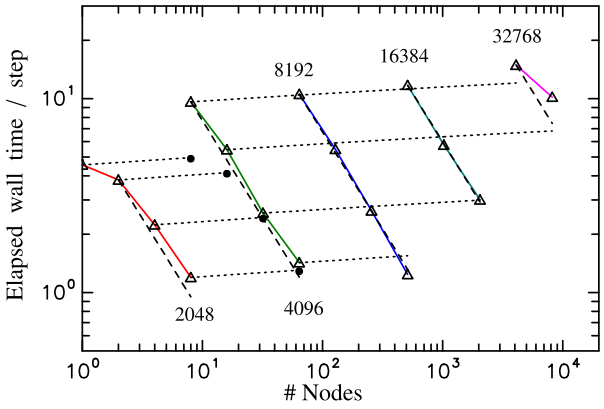


Fig. 7. Plot of overall scalability of the pencils code showing elapsed wall times at different node counts for problem sizes 2048^3, 4096^3, 8192^3, 16384^3 and 32768^3, in different colors. Open triangles show best timings obtained for each combination of N and P. Bullets show theoretical 2048^3 timings if scaled perfectly from 2048^3. Dashed lines of slope -1 on logarithmic scales indicate perfect strong scaling, while perfect weak scaling based on computational workload is represented by dotted lines showing $\log(P^{1/3})$ behavior.

Although for strong scaling we have focused on the pencils code, we also note that the slabs code does perform well in strong scaling. For 8192^3 the wall times per step were 10.72, 5.49 and 2.73 seconds using 64, 128 and 256 nodes respectively; and for 16384^3 they were 11.30, 5.57 and 2.89 seconds using 512, 1024, 2048 nodes respectively, both giving 98% strong scaling for a 4x increase in node count. These numbers are close to results reported for the pencils code in Tables 5 and 6. However, consistent with the limitations stated earlier, for the case of 32768^3 the slabs code can only run at a single node count (4096), so no slabs version of Table 7 is available.

Ultimately, communication across a large number of MPI processes is the primary limiting factor where extreme-scale code performance data are concerned. In fact, in GPU codes, communication may become even more dominant than for CPU codes, as flop-rate increases by a large factor while communication performance may only improve modestly. In the next section, we provide a deeper analysis of communication performance, including the effect of message sizes below an optimal value on the system.

Table 8
Theoretical peak bandwidths corresponding to the possible communication pathways on Frontier. The CPU and GPU Memory Bandwidths are relevant for the self-communication that occurs with each MPI_AlltoAll communication, The bandwidths in the Peak Bandwidth column present the CPU and GPU bandwidths in terms of "*send rate* + *receive rate*" to be consistent with OLCF online documentation.

Communication Pathway	Peak Bandwidth GB/sec
CPU Memory Bandwidth	205
GPU Memory Bandwidth	1635
GCD-to-GCD on same GPU	(200 + 200)
GPU-to-GPU in pairing	(100 + 100)
GPU-to-GPU out of pairing	(50 + 50)
CPU-to-CPU and GPU	(36 + 36)
Node-to-Node per NIC	(25 + 25)

5. Analyis of MPI AlltoAll communications performance

The pre-dominance of MPI all-to-all costs in wall clock timings presented in Sec. 4 merits a deeper discussion into how the actual communication performance achieved in the present work compares with the theoretical peak communication performance available on Frontier, and where hopes for a future breakthrough may lie for codes which are communication intensive in nature. We refer frequently below to Ref. 16 which can be regarded as an official reference on expectations for user application performance given the system characteristics on Frontier.

Several features of the network design on Frontier are also relevant in this discussion. Specifically, each node on Frontier contains 4 AMD MI250x GPUs, each of which contains 2 Graphics Compute Dies (GCDs) for a total of 8 GCDs per node. Per official system information [16], each GCD effectively functions as a separate device. To best utilize Frontier, we always use 8 MPI ranks per node, with one MPI rank per GCD and 2 MPI ranks per GPU. The AMD MI250x GPUs on Frontier are designed with the network interface card (NIC) as an inherent part of the GPU, such that all the MPI ranks and their associated GCDs and CPUs share the NIC for their I/O and communications across the network. This design reduces the number of buses between the device memory and the network compared to host memory, thus lowering latency and potentially increasing bandwidth. Our distribution of MPI ranks across the nodes and GPUs results in 2 MPI ranks sharing each NIC and its available bandwidth. Further, the GPU-Direct MPI communications also benefit from exceedingly fast on-node communication pathways between the GPUs. The benefit of these high-speed on-node connections is evident in the timings for cases performed on a single node or when performed with only a few MPI ranks in a specific MPI communicator (for instance, with row-communicator timings improving with decreasing P_r in the tables shown in Sec. 4).

Frontier employs the Slingshot 11 interconnect in a dragonfly network topology with 128 nodes assigned to each dragonfly group. Slingshot dynamically routes the traffic to reduce congestion with a maximum of 3 "hops" across the network between nodes. The potential communication bandwidths are summarized in Table 8 in terms of "*send rate* + *receive rate*" to be consistent with Ref [16] and OLCF online documentation. For the purposes of our comparisons, we will consider only the uni-directional, receive-side bandwidths to be consistent with current literature and to avoid confusion in the comparisons. The four GPUs on a node are arranged in two pairs. The GPUs within a pairing can communicate at uni-directional rates up to 100 GB/sec (2 links at 50 GB/sec/link) while communications between GPUs in the other pairing are limited to 50 GB/sec (1 link at 50 GB/sec). The GCDs within a GPU have a theoretical peak communication rate of 200 GB/sec (4 links at 50 GB/sec/link) with each other. With the knowledge that each MPI rank is performing the same volume of all-to-all communication, we can safely assume that the available network bandwidth (B/W) is evenly split between the 2 MPI ranks on each GPU for a theoretical peak B/W of 12.5 GB/sec/rank. Further explanations of communication at the systems level are available in Ref. [16].

As stated earlier (Sec. 2.2 and elsewhere) a key decision in our coding development is to store all major arrays on the GPU and effectively perform all computations there. This strategy allows us to minimize data movement between the host and device and provides a strong incentive to communicate between the MPI processes directly from the GPUs. Performing the communications from the CPUs would have required the exchange of data between the host and device before and after the MPI calls, which definitely increases overall runtime. It is interesting to quantify the savings in cost achieved by performing MPI on GPUs instead of on CPUs on Frontier.

Besides timings collected for all-to-all communication calls, another important measure of MPI performance is the effective bandwidth for a given message size and number of participating MPI ranks. When an MPI all-to-all communication is performed between R MPI ranks, each rank completes 1 self-communication with itself, sends $(R-1)$ point-to-point (P2P) messages to the other $(R-1)$ ranks, while also receiving $(R-1)$ P2P messages of the same size from all other $(R-1)$ ranks. Although the self-communication never crosses any of the communication pathways between the ranks and is not a "communication" in the truest sense, the data must be physically copied to the receive buffer to complete the all-to-all communication, and this data copy must occur concurrently with the rest of the all-to-all operations that are being timed (see Fig. 3). Therefore, as in [6], we include the self-communication message in the calculations for an "effective" bandwidth, with the understanding that the self-communication becomes a very small portion of the all-to-all operations when R is large. However, we note that bi-directional bandwidths were quoted in [6] for previous work on Summit. Since we're limiting our comparisons to uni-directional (i.e. receive-side) bandwidths to be consistent with current literature, the effective uni-directional bandwidths per MPI rank is then given by the formula

$$BW = (P2P \ \text{x} \ R) / \ t_{MPI} \tag{13}$$

where $P2P$ represents the peer-to-peer message size, R is the number of MPI ranks in the communicator, and t_{MPI} is the elapsed wall time measured for the MPI_ALLTOALL call concerned.

Two issues addressed in Ref. [16] affect our bandwidth measurements significantly. First, the efficiency of the MPI communications degrades as the size of the P2P messages decreases, which adversely affects the performance of our larger runs. As explained earlier in Sec 4.3, as the problem size increases, the overall communication volume per MPI rank remains constant while the individual P2P message sizes decrease. Not only does this decrease result in lower communication performance due to an inability to saturate the MPI communications fully with data to transfer, but it also renders the code more susceptible to communication variability and leads to a greater impact of the MPI latencies on the collected timings. The second issue is the limitation on the usable network bandwidth imposed by the dragonfly topology for communications between nodes in different dragonfly groups. Ref. [16] reports that communications between nodes in the same dragonfly group achieved a receive-side bandwidth of 17.5 GB/sec/NIC out of the possible 25 GB/sec/NIC, but communications between nodes in different dragonfly groups were limited to 3 to 8 GB/sec/NIC. Since 2 MPI ranks share a single NIC in our cases, we halve these rates for the following discussions: 8.75 GB/sec/rank for in-group communications and 1.5 to 4 GB/sec/rank for out-of-group communications. With these issues in mind, the effective bandwidths at larger problem sizes are not as high compared to the theoretical peak rates (in Table 8) as we would like, but they are on the higher end of the ranges discussed in Ref. [16].

To aid our discussions, the theoretical peak communication bandwidths for the various communication patterns used in the slab (1D) and pencil (2D) decomposition codes are listed in Table 9. All five rows in

Table 9
Theoretical uni-directional bandwidths for communications between GPUs with P MPI ranks in the communication. Communications for $P = 4$ may occur between GPUs not-in-pairing at 50 GB/sec or between GPUs in-pairing at 100 GB/sec.

Number of Ranks P	Expected B/W GB/sec
1	1635
2	200
4	50 or 100
8	50
more than 8	12.5

the table are pertinent to our 2D decompositions on Frontier, while only the last two rows apply to 1D decompositions. The first four rows give bandwidths expected for on-node communications while the last row gives bandwidth expected for node-to-node communications. In general, many communication calls will involve transfers along different pathways of different speeds. Since MPI_ALLTOALL is a blocking call, the timing measured by MPI_WTIME calls placed before and after MPI_ALLTOALL is determined by the slowest link involved in the communication. This is relevant in our quest for a detailed explanation of the timings, as below.

To collect MPI performance data needed for our purposes here, we ran a lightweight MPI all-to-all program which uses communication patterns identical to those in the fully-featured DNS codes but perform strictly MPI_ALLTOALLs only, without any other operations. All of the MPI timings presented below are the average of the timings of 5 separate and complete communication cycles that were conducted sequentially. It should be noted that timings associated with MPI on the CPU include the extra cost of data transfer between the host and device, which would be necessary since all computations will continue to be performed on the GPUs. To minimize any contamination from network contention, we use both CPU and GPU timings from the same executable and the same job script. Timings extracted from our MPI program show very little variability, perhaps because MPI calls performed by this program are often occurring within only a couple of seconds apart, in contrast to those in full DNS codes where data from successive time steps span a longer time window and may thus be more susceptible to unpredictable fluctuations in the level of network activity on the machine. We present MPI performance data on a per-variable basis for 1D and 2D decompositions separately.

5.1. 1D decomposition: slabs

Table 10 shows data from the slabs code on MPI alltoall message size, timings on both CPU and GPU, and effective bandwidth attained on the GPU for a single variable at five problem sizes. The CPU timing data includes time taken for data movement between the host and the device. As expected, the actual MPI communications from the GPU are faster than those from the CPU since the NIC is on the GPU, and the data has a longer and slower pathway to the NIC, and the network beyond, compared to the GPU [16]. In addition to slower communications, performing the MPI_ALLTOALL call from the CPUs is further burdened by the requisite H2D and D2H data movement —which, as seen in the table — costs more than the actual MPI communications themselves. The overall cost of performing the communication on the CPUs is 2.4x of communication on the GPU, for the largest problem size on 4096 nodes. This difference also becomes even more significant for smaller problems ranging from 3.65x on 512 nodes to 13x on a single node.

As noted earlier, the single-node case benefits from the high-speed on-node pathways that connect the GPUs while the majority of the communication in cases using multiple nodes must go node-to-node across the network at a much lower data transfer rate. Taking the single-node cases as an example, the 8 MPI ranks communicate across multiple on-node links: 1 self-communication, 1 GCD-to-GCD communication within a GPU, 2 communications with GCDs with the other GPU in pairing, and 4 communications with GCDs in the other GPU-pairing which are the slowest links in this communication. Hence, we expect a uni-directional peak rate of 50 GB/sec for the single-node cases as shown in Table 9 (fourth row). In contrast, for the 8-node cases with 64 MPI ranks, we have the aforementioned 8 communications as well as 56 additional communications with GCDs in other nodes, and it's these node-to-node communications that limit the expected peak rate to 12.5 GB/sec (Table 9). Indeed, the effective GPU-aware MPI bandwidth drops by roughly a factor of 4 between the first two rows of Table 10, which is consistent with the theoretical node-to-node bandwidth being 1/4 of the on-node bandwidth for 8 MPI ranks.

For $N = 2048$ on a single node, the effective bandwidth was measured to be 32.9 GB/sec. This rate is 65.8% of the expected 50 GB/sec, and slightly lower than the 72% to 75% of theoretical peak achieved by [16]. However, Ref. [16] is not clear on what kind of communications were used in their tests (i.e. individual P2P or collective), and 32.9 GB/sec may be reasonable for our collective communications.

As more and more of the MPI communications go across the network as N increases, the network bandwidth becomes increasingly dominant in the performance of the communications. Reasonable weak scaling is seen from $N = 4096$ to $N = 16384$, but a modest drop occurs at $N = 32768$. As mentioned earlier, for a fixed volume of communication per MPI rank, increasing the number of MPI ranks as the problem size grows necessarily decreases the P2P message sizes that must be distributed across all the ranks. Eventually, the small P2P message sizes degrade the communication performance (Ref. [16]), and this is particularly true for our largest case, $N = 32768$. For $N = 32768$, the individual P2P message size drops to 0.125 MiB, which when combined with a large collective communication, results in a failure to saturate MPI fully with data to transfer, leading to a slight reduction of performance. This situation is not surprising, since although network hardware and MPI implementation vary, most massively parallel codes scale less well at the extreme limits of capacity on most large machines (e.g., [43]). This situation is further complicated by the limitations in usable network bandwidth imposed by the dragonfly topology. Reference [16] observed a range of receive-side communication rates from 1.5 to 4 GB/sec/rank on Frontier for communications that must go between dragonfly groups, and we see that all effective bandwidths for multi-node cases in Table 10 exceed these rates. This is most important for the two largest cases on 512 nodes and 4096 nodes since they naturally have the smallest P2P message sizes and these node counts necessarily span multiple dragonfly groups. Although cases using up to 128 nodes can be placed in a single dragonfly group, this is not guaranteed when the system is highly utilized, and even an 8-node case could be spread across multiple dragonfly groups. Our effective bandwidths for cases using 8 or more nodes exceed the upper range of 4 GB/sec/rank that is presented in [16].

Although slower than communications from the GPU, the actual MPI communications from the CPUs perform very consistently and, for the most part, behave as expected as the problem size increases in Table 10. The CPU MPI communication for $N = 16384$ on 512 nodes appear to be somewhat out of line, compared to both using CPUs for $N = 32768$ on 4096 nodes and using GPUs for $N = 16384$ on 512 nodes. This observation is repeatable and not an artifact of variability. However, this has no impact on our work since all production simulations are performed using GPU MPI communications which consistently provide better performance.

5.2. 2D decomposition: pencils

For the 2D decomposition strategy, two communication cycles must be completed for each forward and inverse 3D FFT computation. These

Table 10
A comparison between MPI using host memory pointers and MPI using device memory pointers, in terms of communication timings (in seconds), message sizes and effective uni-directional bandwidth per MPI rank on Frontier, for 1D decomposition for a single variable. Since the numbers of nodes and MPI ranks are chosen in proportion to N^3, the communication volume is fixed at 4 GiB (i.e. 4096 MiB) in all cases shown.

N^3	Nodes	Ranks	P2P Msg Size (MiB)	H2D/D2H Copy (sec)	CPU MPI (sec)	CPU Total (sec)	GPU MPI (sec)	GPU B/W (GB/sec)
2048^3	1	8	512	1.256	0.450	1.706	0.131	32.90
4096^3	8	64	64	1.213	0.687	1.900	0.494	8.71
8192^3	64	512	8	1.204	0.744	1.948	0.530	8.10
16384^3	512	4096	1	1.187	0.966	2.152	0.590	7.28
32768^3	4096	32768	0.125	1.210	0.796	2.006	0.771	5.58

Table 11
A comparison between the row-communicator MPI on the CPU versus on the GPU with communication timings (in seconds), including information on message sizes, and effective uni-directional bandwidth per MPI rank on Frontier for a single variable. Since the numbers of nodes and MPI ranks are chosen in proportion to N^3, the row communication volume is fixed at 4 GiB (i.e. 4096 MiB) in all cases shown.

N^3	Nodes	P_r	P2P Msg Size (MiB)	H2D/D2H Copy (sec)	CPU MPI (sec)	CPU Total (sec)	GPU MPI (sec)	GPU B/W (GB/sec)
2048^3	1	2	2048	1.239	0.393	1.632	0.046	92.98
4096^3	8	2	2048	1.184	0.392	1.576	0.046	92.97
8192^3	64	2	2048	1.192	0.392	1.584	0.046	93.05
16384^3	512	2	2048	1.181	0.392	1.573	0.046	93.13
32768^3	4096	2	2048	1.173	0.393	1.566	0.046	92.88

Table 12
Similar to Table 11, but for column communicators. The column communication volume is also fixed at 4 GiB (i.e. 4096 MiB) in all cases shown.

N^3	Nodes	P_c	P2P Msg Size (MiB)	H2D/D2H Copy (sec)	CPU MPI (sec)	CPU Total (sec)	GPU MPI (sec)	GPU B/W (GB/sec)
2048^3	1	4	1024	1.241	0.444	1.685	0.092	46.93
4096^3	8	32	128	1.185	0.660	1.845	0.539	7.97
8192^3	64	256	16	1.224	0.689	1.913	0.580	7.41
16384^3	512	2048	2	1.223	0.826	2.049	0.626	6.87
32768^3	4096	16384	0.25	1.231	0.848	2.079	0.705	6.09

two cycles are identified as "row" and "column" communications in our discussions. We compare the performance of MPI on the CPU and GPU for a single variable for five problem sizes in the row and column communicators separately in the tables below. As with the 1D decomposition timings, the timings given for MPI on the CPUs include a substantial contribution from the device-to-host and host-to-device data movements that occur respectively before and after an MPI_ALLTOALL on the CPUs. In addition to assessing MPI performance at a basic level, the MPI timings also provide an interesting check on the communication costs cited in Tables 4 to 7 if the number of MPI calls of each type performed per DNS time step is taken into account. As discussed in Sec. 4, GPU performance data for the pencils code are dependent on the shape of the 2D processor grid, as well as the number of variables passed per communication call. For the sake of brevity and unless stated otherwise, we consider only the case $P_r = 2$ and $Q = 1$ here.

Table 11 shows row-communicator performance data. Because the number of nodes is weak-scaled to the problem size, the P2P message sizes sent or received are equal to 2 GiB in each case shown in the table. For the CPU, The cost of H2D or D2H data movement is about 3x that of the actual transfer. On the GPU, with $P_r = 2$, the two MPI ranks in each row communication are exchanging data between the GCDs on a single GPU across the fastest on-node pathway available on Frontier. As a result, the GPU MPI row communication is very fast — in fact, over 30x faster than using the CPU. However, the effective bandwidth of nearly 93 GB/sec can be a bit misleading since the self-communication is included in our bandwidth computations, and for $P_r = 2$, the self-communication is half of the data being exchanged (at a theoretical peak bandwidth of 1635 GB/sec). Unfortunately, our timing strategy (see Fig. 3) does not allow us to analyze the performance of the individual P2P operations generated by each MPI_ALLTOALL function call. Instead, we can only evaluate the performance of the overall MPI_ALLTOALL function, and therefore, we include the self-communication in the bandwidth computations. With the knowledge that the GCD-to-GCD link is the slowest pathway in the communication for $P_r = 2$ (e.g. 200 GB/sec vs 1635 GB/sec), we can make some additional observations. Since only half of the data actually crosses the links between the two GCDs for $P_r = 2$, the bandwidth of data actually being exchanged between the GCDs is only 46.5 GB/sec which equates to 23.25% of the expected uni-directional peak bandwidth of 200 GB/sec. Reference [44] explains that Cray-MPICH cannot fully utilize the 4 links (50 GB/sec/link) between the two GCDs in a GPU due to limitations with the Direct Memory Access (DMA) copy engine and reports a maximum receive-side (i.e. uni-directional) bandwidth of 50 GB/sec in their tests. This rate is identical to the System Direct Memory Access (SDMA) data transfers between GCDs that are discussed in Ref. [16] which attributes the bandwidth limitation to the inability to stride the SDMA transfers across multiple links. With these limitations in mind, we are communicating at 93% of peak bandwidth across a single link between the GCDs, and our communications are consistent with current literature. Additionally, since all these row communications occur on-node, both the CPU and GPU communications experience low congestion and are completed with very high consistency. The number of row communicator MPI calls that need to be made during each time step is (if $Q = 1$) 16. It can be seen that multiplying the GPU timing of 0.046 secs by 16 gives a very close match with 0.74 seconds for $P_r = 2$ in the pencils-code tables discussed in Sec. 4.

Table 12 presents the timings for the 2D decomposition column communications. For the single-node case, each column communication necessarily occurs between GCDs in all four GPUs in the node due to our decomposition algorithm, and this limits the expected peak bandwidth to 50 GB/sec. Our effective bandwidth of 46.93 GB/sec achieves 93.9%

Table 13
A comparison of the GPU MPI communication uni-directional bandwidths per MPI rank when passing 1 variable ($Q = 1$) and 4 variables ($Q = 4$) for $N^3 = 16384^3$ on 2048 nodes and $N^3 = 32768^3$ on 8192 nodes.

N^3	Nodes	P_r	P_c	Q	Row			Column		
					P2P Msg Sze (MiB)	MPI (sec)	B/W (GB/sec)	P2P Msg Size (MiB)	MPI (sec)	B/W (GB/sec)
16384^3	2048	8	2048	1	128	0.0298	36.04	0.500	0.171	6.275
				4	512	0.1175	36.57	2.000	0.652	6.590
		4	4096	1	256	0.0225	47.64	0.250	0.167	6.450
				4	1024	0.0896	47.94	1.000	0.660	6.505
		2	8192	1	512	0.0115	93.04	0.125	0.275	3.901
				4	2048	0.0462	92.91	0.500	0.684	6.275
32768^3	8192	8	8192	1	256	0.0592	36.27	0.250	0.536	4.006
				4	1024	0.2339	36.72	1.000	1.959	4.385
		4	16384	1	512	0.0450	47.73	0.125	0.491	4.371
				4	2048	0.1789	48.03	0.500	1.848	4.648
		2	32768	1	1024	0.0232	92.50	0.0625	0.725	2.960
				4	4096	0.0924	92.94	0.250	2.212	3.884

of the peak rate. In addition to the excellent single-node performance, the effective bandwidths of the other four cases, using 8 or more nodes, exceed the upper range of 4 GB/sec/rank that is presented in [16]. Altogether, these results demonstrate a very high level of efficiency for these communications.

For the single-node case of 2048^3 the effective bandwidth is about half of that seen for the row communicator. In contrast to the row communicator, the P2P message size drops by a factor of 8 for each doubling of N, which causes a reduction in effective bandwidth consistent with the DNS timing trends examined earlier. Multiplication of GPU MPI timings here by a factor of 14 gives a very close match with column-communicator timings seen with $P_r = 2$ in Sec. 4.2, provided variability due to contention does not arise in DNS timings used for the comparison.

It may be noted that the timings in Table 12 also show behaviors that are very similar to and slightly faster than the 1D decomposition timings presented in Table 10. This similarity is expected since these communications are performed in nearly identical manners with one key difference: with $Pr = 2$ and hence $P_c = P/2$, the column communicators involve half the number of MPI ranks than the 1D decomposition communicators and this results in their P2P message sizes being twice the size of their 1D decomposition counterparts. It is likely that the faster performance and higher bandwidth of the 2D decomposition column communications is related to the fact that their larger P2P message sizes better saturate the communications with data to transfer.

We noted earlier (Sec 4) that increasing the message size by changing the number of variables (Q) to be transposed through each MPI_ALLTOALL call from 1 to 4 leads to more consistent performance for $N = 16384$ but has lesser impact on $N = 32768$. A deeper investigation into the differences in behaviors between these two problem sizes illustrates the impacts of both the P2P message sizes and the dragonfly topology on problems that are distributed across more nodes and consequently more dragonfly groups. Table 13 presents the effect of message size on the MPI communications and mimics the decomposition and communication patterns of the cases using 2048 nodes in Table 6 and 8192 nodes in Table 7. References [16] and [44] discuss a gradual degradation in MPI communication performance as the P2P message sizes fall below 1 MB, and the column communications for $Q = 1$ (and smaller P2P message sizes) are slower than those for $Q = 4$ for all choices of P_r. Further, $P_r = 2$ with the smallest P2P message size for the column communications are clearly slower than the other two choices for P_r. However, the column communications for $P_r = 4$ are faster than those for $P_r = 8$ even though the P2P message sizes for $P_r = 4$ are smaller than those for $P_r = 8$. This behavior is also seen between the respective rows in Table 7 for 8192 nodes and also in Table 3 for $N = 32768$ on 4096 nodes for $P_r = 2$, $P_r = 4$, $P_r = 8$. The trend highlights a dependence upon the choice of decomposition in addition to the P2P message size and the dragonfly topology. This work is the first known demonstration of this behavior. Regardless of the choice of P_r, the column communication bandwidths either exceed or are in the upper half of the range of usable network bandwidths (1.5 to 4 GB/sec/rank) for the dragonfly group-to-group communications observed in Ref. [16].

All the row communications in Table 13 occur within a node, and consequently demonstrate high bandwidths similar to other single-node cases. The behavior of the row communications for $P_r = 2$ is consistent with that seen in Table 11 and no further discussion is needed. The row communications with $P_r = 4$ occur between GPUs in a pairing with a uni-directional ceiling of 100 GB/sec while those for $P_r = 8$ occur between GPUs both within and outside the pairing and therefore have a ceiling of 50 GB/sec. The row communications for $Pr = 8$ achieve upwards of 72% of the theoretical peak which is consistent with [16]. Although they have a theoretical peak of 100 GB/sec, we postulate that the in-pairing communications for $P_r = 4$ exhibit similar bandwidth limitations (i.e. 50 GB/sec) as experienced by the GCD-to-GCD communications for $P_r = 2$ due to the current implementation of the copy engines ([16] and [44]). However, the complexity in the connections and in how the communications are managed by Slingshot make it difficult to explain fully at this point. Nevertheless, even at these larger P2P message sizes, the communications do benefit slightly from larger message sizes.

A cross-check between communication performance in the slabs and pencils codes can be made by applying a 1D decomposition strategy within the 2D decomposition MPI kernel. Table 14 presents timings for the pencils code when used in a so-called "slabs mode", with $P_r = 1$. While operating in slabs mode, the 2D decomposition code operates much like the 1D decomposition code in that the domains are partitioned in identical manners and only one MPI communication cycle is needed. All MPI communications occur through the column communications, while the row communications are replaced with simple data copies to place the data in the required format for the FFT calculations in the second direction of the slab. As expected, these timings compare well with those in Table 10.

6. Conclusions

The arrival of the world's first exascale computer [15], which remains the world's fastest supercomputer at the time of preparation of this paper, has generated much renewed optimism for the next frontier of computational science in many technical disciplines. It is also clear that substantial investments of algorithmic and software effort are essential for user applications to attain their fullest possible potential in fully exploiting the advanced GPU and network technologies provided on the most powerful system, *Frontier*, at Oak Ridge National Laboratory, USA.

Table 14
Similar to Table 12, but for configurations with $P_r = 1$.

N^3	Nodes	P_c	P2P Msg Size (MiB)	H2D/D2H Copy (sec)	CPU MPI (sec)	CPU Total (sec)	GPU MPI (sec)	GPU B/W (GB/sec)
2048^3	1	8	512	1.249	0.466	1.715	0.118	36.55
4096^3	8	64	64	1.216	0.667	1.883	0.494	8.695
8192^3	64	512	8	1.229	0.729	1.958	0.582	7.380
16384^3	512	4096	1	1.243	0.955	2.198	0.595	7.225
32768^3	4096	32768	0.125	1.244	0.851	2.095	0.651	6.595

This paper represents the outcome of such an effort, motivated primarily by the complexity of fluid turbulence. However, we believe the lessons learned will be relevant also to other user communities having an interest in multi-dimensional distributed-memory Fast Fourier Transforms (FFTs), or other numerical techniques whose scalability at the extreme limits is ultimately constrained by communication costs in pursuit of massive parallelism on either CPUs or GPUs.

Our direct objective is to develop a capability for simulating 3D turbulence at the highest grid resolution that the memory capacity of Frontier can support, which also exceeds the scale of prior efforts in the field. For us, the most important characteristics of Frontier, comprised of AMD MI250x nodes, are that the CPU and GPU memory are of the same size and the network cards are an integrated component of the GPU itself. These features allow us to perform effectively the entire computation, except perhaps for input and output, on the GPUs, while simplifying the coding via a one-to-one correspondence between MPI ranks and GCDs. We use OpenMP offloading on the GPU, with the entire time-stepping loop enclosed within a giant OMP TARGET DATA MAP construct, thus avoiding the cost of copying between the CPU (host) and GPU (device) almost completely. All-to-all communication is performed directly on the GPU, which is faster and also free of extra cost in host-device copying that would be incurred if using CPUs. Our strategy here is quite distinct from that previously used [6] on Summit, which has a different GPU-to-CPU memory ratio. Although in our code the CPUs are left largely idle, avoiding host-device copying on Frontier is much more beneficial than distributing work to the CPUs which are of much lower computational speed.

Several key programming principles adopted in this work are illustrated via code fragments in Fig. 3–6. The 3D solution domain with N^3 grid points is divided among P parallel processes, into slabs or pencils using 1D and 2D domain decompositions respectively. In both cases, careful management of the memory footprint is important (Eq. (9)). The slabs code is simpler, and faster at some problem sizes. However, the pencils code does provide comparable performance, and is more flexible in allowing $P > N$ if more memory is needed than when solving for just the velocity field, or if an aggressive reduction in time-to-solution is desired.

A systematic investigation of code performance has been conducted at several problem sizes, ranging from 2048^3 (which fits on a single node) to 32768^3 (which requires at least 4096 nodes). Weak scaling tests are conducted by letting $P \propto N^3$ while taking computational cost to be proportional to $N^3 \log_2 N$ based on 3D FFT operation counts. As an artifact of MPI performance on the node being much faster than across multiple nodes, weak scaling is low between 2048^3 on 1 node and 4096^3 on 8 nodes, but consistently high at larger problem sizes and node counts. The slabs code provides a factor of 6 advance in figure of merit based on number of grid points per second for 32768^3 on 4096 nodes on Frontier compared to 18432^3 [6] on 3072 nodes on the IBM-NVIDIA machine "Summit" at the OLCF. The pencils code is (Table 4) also about 10x faster than a corresponding CPU-only version on Frontier.

To understand the performance, we have used detailed internal profiling to identify separate contributions to elapsed wall time per time step, from 1-D FFTs (performed using the AMD ROCm library), packing and unpacking operations before and after all-to-all communication, the all-to-all calls themselves, and other operations (including forming nonlinear products, advance in time, etc). Most of the performance data in the paper (Tables 3, 5, 6, 7) relate to the pencils code, whose performance also depends on the shape of the 2D processor grid ($P_r \times P_c$, as dimensions for row and column sub-communicators). We choose P_r to be an integer factor of the number of GCDs accessible on each node (8) so that the MPI in the row-communicator can benefit from high on-node bandwidth. In the case of $P_r = 1$ further savings are obtained by avoiding self-communication altogether. However, a challenge does arise at the extreme problem size of 32768^3, where a small P_r leads to a large P_c and hence smaller message sizes in the column communicator, which can result in lower effective bandwidth and significant performance variability.

While weak scaling is closely related to the problem size and ultimate science impact of the simulations, good strong scaling is also important in attempts to shorten the time to solution at larger problem sizes. Table 5 shows excellent strong scaling for 8192^3 up to 8x increase in node count versus the minimum required for memory. Such excellent strong scaling is contingent upon perfect scalability of MPI_ALLTOALL, which is more difficult at larger problem sizes with higher node counts and smaller message sizes. Table 6 shows timings at 16384^3 are still very good up to 4x increase in node count, but only when the issue of small message size in the column communicator is alleviated by making MPI all-to-all communication calls with up to 4 variables at a time. However, ultimately at 32768^3 (Table 7) the strategy of multi-variable all-to-all produces only a lesser improvement.

The connections between MPI performance characteristics and the Frontier machine architecture have been given more attention in Sec. 5. It is shown explicitly that the overhead of host-device copying is a significant contribution to MPI on the CPU being slower than MPI on the GPU. We have verified that effective bandwidth is very high in communications between a pair of GCDs, becomes slower if taken over the entire node, and even slower if taken across multiple nodes on the system. The difference in observed bandwidth between the row and column communicators is also drastic, especially when the message sizes in the latter drop as $P_c \propto N^3$ with P_r held fixed. In addition, the dragonfly network topology on Frontier has the effect of making certain inter-node communications travel along more complex pathways, leading to less efficiency even for the same message size as the node count increases. The effective bandwidths measured in this code are consistent with official expectations discussed in [16].

Taking all factors into account, we believe the overall picture of scalability as illustrated in Fig. 7 is very favorable. There, the only data points straying significantly off the asymptotes representing theoretical limits of perfect weak or strong scaling, are those for 2048^3 at 1 or 2 nodes (an artifact of very fast intra-node communication), and those for 32768^3 pushing against the boundaries of what is possible on the machine. Considering that a large speedup of computational parts in GPU codes actually makes the MPI more dominant which then often leads to lower scalability, the present performance results are in fact arguably better than could have been expected, with the strengths of the Slingshot interconnect playing an important role. From our perspective, future improvements in interconnect performance, including raw bandwidth and reduction of variability, are likely to be among the most desired features for the next generation(s) of leadership-class supercomputers to come.

Finally, we note that the pencils code described in this paper has already been used for some production simulations focusing on the study of intermittency in turbulence at Taylor-scale Reynolds number close to 2500, at resolution up to 32768^3 to be addressed in an upcoming

paper. The same code is also used as the underlying framework for tracking fluid particles in the flow for investigations of turbulence from a Lagrangian perspective [45] and of the turbulent transport of passive scalars with different computational requirements for regimes of low versus moderate molecular diffusivity [46,47]. Ultimately, the science impact of 32768^3 simulation data enabled by the present work will be amplified further when selected datasets are ingested into the Johns Hopkins Turbulence database [48] for public access, and felt in a variety of important areas such as those typified by Refs. [49,50].

CRediT authorship contribution statement

P.K. Yeung: Writing – original draft, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Conceptualization. **Kiran Ravikumar:** Writing – review & editing, Software, Methodology, Formal analysis, Conceptualization. **Stephen Nichols:** Writing – original draft, Software, Resources, Methodology, Formal analysis, Conceptualization. **Rohini Uma-Vaideswaran:** Writing – review & editing, Methodology, Investigation, Data curation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgements

The authors gratefully acknowledge support from the OLCF CAAR program from 2020 to 2023 and DOE INCITE Awards in years 2023 and 2024. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

The work at Georgia Tech is sustained by a subaward from NSF via CSSI Grant 2103874 led by C. Meneveau of The Johns Hopkins University. For strong encouragement and valuable technical advice we are indebted to many current or former members of OLCF, HPE and AMD staff, including (in alphabetical order) Steve Abbott, Alessandro Fanfarillo, Oscar Hernandez, John Levesque, Nick Malaya, Bronson Messer, Mark Stock, Matthew Turner, and Jack Wells. PKY also acknowledges the impetus for work on turbulence from enduring science collaborations with Toshiyuki Gotoh, Charles Meneveau, Stephen B. Pope, and Katepalli R. Sreenivasan. Finally, we thank two anonymous reviewers for their constructive comments.

References

[1] S.B. Pope, Turbulent Flows, Cambridge University Press, Cambridge, U.K, 2000.
[2] K.R. Sreenivasan, Fluid turbulence, Rev. Mod. Phys. 71 (1999) s383–s395.
[3] Y. Zhou, Turbulence theories and statistical closure approaches, Phys. Rep. 935 (2021) 1–117.
[4] M. Yokokawa, T. Itakura, A. Uno, T. Ishihara, 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, 2002.
[5] M. Lee, N. Malaya, R.D. Moser, Petascale direct numerical simulation of turbulent channel flow on up to 786 K cores, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, ACM, Denver, Colorado, USA, 2013, pp. 61:1–61:11.
[6] K. Ravikumar, D. Appelhans, P.K. Yeung, GPU acceleration of extreme scale pseudo-spectral simulations of turbulence using asynchronism, in: Proceedings of the International Conference for High Performance Computing, Networking and Storage Analysis (SC'19), Denver, CO, USA, ACM, New York, NY, USA, 2019.
[7] P. Moin, K. Mahesh, Direct numerical simulation: a tool in turbulence research, Annu. Rev. Fluid Mech. 30 (1998) 539–578.
[8] S. Hoyas, J. Jimenez, Scaling of the velocity fluctuations in turbulent channels up to $Re_\tau = 2003$, Phys. Fluids 18 (2006) 011702.
[9] T. Ishihara, T. Gotoh, Y. Kaneda, Study of high-Reynolds number isotropic turbulence by direct numerical simulation, Annu. Rev. Fluid Mech. 41 (2009) 165–180.
[10] M. Lee, R.D. Moser, Direct numerical simulation of turbulent channel flow up to $Re_\tau \approx 5200$, J. Fluid Mech. 774 (2015) 395–415.
[11] P.K. Yeung, X.M. Zhai, K.R. Sreenivasan, Extreme events in computational turbulence, Proc. Natl. Acad. Sci. 112 (41) (2015) 12633–12638.
[12] T. Ishihara, K. Morishita, M. Yokokawa, A. Uno, Y. Kaneda, Energy spectrum in high-resolution direct numerical simulation of turbulence, Phys. Rev. Fluids 1 (2016) 082403.
[13] T. Watanabe, J.J. Riley, S.M. de Bruyn Kops, P.J. Diamessis, Q. Zhou, Turbulent/non-turbulent interfaces in wakes in stably stratified fluids, J. Fluid Mech. 797 (2016) R1.
[14] P.K. Yeung, K. Ravikumar, Advancing understanding of turbulence through extreme-scale computation: intermittency and simulations at large problem sizes, Phys. Rev. Fluids 5 (2020) 110517.
[15] R. Budiardja, M. Berrill, M. Eisenbach, G. Jansen, W. Joubert, D.S. Nichols, D. Rogers, A. Tharrington, B. Messer, Ready for the frontier: preparing applications for the world's first exascale system, https://doi.org/10.1007/978-3-031-32041-5, 2023, pp. 182–201.
[16] S. Atchley, C. Zimmer, J. Lange, D. Bernholdt, G. Dicha, T. Beck, M.J. Brim, R.D. Budiardja, S. Chandrasekaran, M. Eisenbach, T. Evans, M. Ezell, N. Frontiere, A. Georgiadou, J. Glenski, P. Grete, S. Hamilton, J. Holmen, A. Huebl, D. Jacobson, W. Joubert, K. Mcmahon, A. Merzari, S. Moore, A. Myers, D.S. Nichols, S. Oral, T. Papatheodore, D. Perez, D.M. Rogers, E. Schneider, J.-L. Vay, P. Yeung, Frontier: exploring exascale, in: SC '23: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Article 52, Denver, CO, USA, ACM, New York, NY, USA, 2023, pp. 1–16.
[17] N. Malaya, B. Messer, J. Glenski, A. Georgiadou, J. Lietz, K. Gottiparthi, M. Day, J. Chen, J. Rood, L. Esclapez, J. White, G. Jansen, N. Curtis, D.S. Nichols, J. Kurzak, N. Chalmers, C. Freitag, P. Bauman, A. Fanfarillo, R.D. Budiardja, T. Papatheodore, N. Frontiere, D. Mcdougall, M. Norman, S. Sreepathi, P. Roth, D. Bykov, N. Wolfe, P. Mullowney, M. Eisenbach, M. Frahan, W. Joubert, Experiences readying applications for exascale, in: SC '23: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Article 53, Denver, CO, USA, ACM, New York, NY, USA, 2023, pp. 1–13.
[18] C. Canuto, M.Y. Hussaini, A. Quarteroni, T.A. Zang, Spectral Methods in Fluid Dynamics, Springer-Verlag, 1988.
[19] A.N. Kolmogorov, The local structure of turbulence in an incompressible fluid with very large Reynolds numbers, Dokl. Akad. Nauk SSSR 30 (1941) 301–305, see also in: J.C.R. Hunt, O.M. Phillips, D. Williams (Eds.), Turbulence and Stochastic Processes: Kolmogorov's Ideas 50 Years on, Royal Society, London, 1991.
[20] A.N. Kolmogorov, A refinement of previous hypotheses concerning the local structure of a viscous incompressible fluid, J. Fluid Mech. 13 (1962) 82–85.
[21] R.N. Bracewell, The Fourier Transform and Its Applications, 3rd edition, McGraw Hill, 2000.
[22] D. Pekurovsky, P3dfft: a framework for parallel computations of Fourier transforms in three dimensions, SIAM J. Sci. Comput. 34 (4) (2012) C192–C209.
[23] P.D. Mininni, D. Rosenberg, R. Reddy, A. Pouquet, A hybrid MPI–OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence, Parallel Comput. 37 (6–7) (2011) 316–326.
[24] A.G. Chatterjee, M.K. Verma, A. Kumar, R. Samtaney, B. Hadri, R. Khurram, Scaling of a fast Fourier transform and a pseudo-spectral fluid solver up to 196608 cores, J. Parallel Distrib. Comput. 113 (2018) 77–91.
[25] A. Ayala, S. Tomov, A. Haidar, J. Dongarra, Heffte: highly efficient fft for exascale, in: International Conference on Computational Science (ICCS 2020), Amsterdam, Netherlands, 2020.
[26] J.A. Turner, J. Belak, N. Barton, M. Bement, N. Carlson, R. Carson, S. DeWitt, J.-L. Fattebert, N. Hodge, Z. Jibben, W. King, L. Levine, C. Newman, A. Plotkowski, B. Radhakrishnan, S.T. Reeve, M. Rolchigo, A. Sabau, S. Slattery, B. Stump, Exaam: metal additive manufacturing simulation at the fidelity of the microstructure, Int. J. High Perform. Comput. Appl. 36 (1) (2022) 13–39.
[27] M. Pharr, R. Fernando, GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems), Addison-Wesley Professional, 2005.
[28] K. Kumari, E. Cleary, D. Desai, D.A. Donzis, J.H. Chen, A. Aditya, Evaluation of finite difference based asynchronous partial differential equations solver for reacting flows, J. Comput. Phys. 477 (2023) 111906.
[29] H.W. Muer, E. Strohmaier, J. Dongarra, H. Simon, M. Meuer, Top500 list, TOP 500, The List. 228.
[30] R. van der Pas, E. Stotzer, C. Terboven, Using OpenMP — The Next Step. Affinity, Accelerators, Tasking, and SIMD, MIT Press, 2017.
[31] M. Klemm, B.R. de Supinski, OpenMP Application Programming Interface Specification Version 5.0, OpenMP Architecture Review Board, 2018.
[32] S. Bak, C. Bertoni, S. Boehm, R. Budiardja, B.M. Chapman, J. Doerfert, M. Eisenbach, H. Finkel, O. Hernandez, J. Huber, S. Iwasaki, V. Kale, P.R. Kent, J. Kwack, M. Lin, P. Luszczek, Y. Luo, B. Pham, S. Pophale, K. Ravikumar, V. Sarkar, T. Scogland, S. Tian, P. Yeung, OpenMP application experiences: porting to accelerated nodes, Parallel Comput. 109 (2022) 102856.

[33] G. Boffetta, Two-dimensional turbulence, Annu. Rev. Fluid Mech. 44 (2012) 427–451.
[34] T. Gotoh, Y. Watanabe, Y. Shiga, T. Nakano, E. Suzuki, Statistical properties of four-dimensional turbulence, Phys. Rev. Fluids 75 (2007) 016310.
[35] O. Ayala, L.-P. Wang, Parallel implementation and scalability analysis of 3d fast Fourier transform using 2d domain decomposition, Parallel Comput. 39 (1) (2013) 58–77.
[36] B. Chapman, B. Pham, C. Yang, C. Daley, C. Bertoni, D. Kulkarni, D. Oryspayev, E. D'Azevedo, J. Doerfert, K. Zhou, K. Ravikumar, M. Gordon, M.D. Ben, M. Lin, M. Alkan, M. Kruse, O. Hernandez, P.K. Yeung, P. Lin, P. Xu, S. Pophale, T. Sattasathuchana, V. Kale, W. Huhn, Y.H. He, Outcomes of OpenMP Hackathon: OpenMP application experiences with the offloading model (Part I), in: S. McIntosh-Smith, B.R. de Supinski, J. Klinkenberg (Eds.), OpenMP: Enabling Massive Node-Level Parallelism, Springer International Publishing, Cham, 2021, pp. 67–80.
[37] B. Chapman, B. Pham, C. Yang, C. Daley, C. Bertoni, D. Kulkarni, D. Oryspayev, E. D'Azevedo, J. Doerfert, K. Zhou, K. Ravikumar, M. Gordon, M. Ben Del, M. Lin, M. Alkan, M. Kruse, O. Hernandez, P.K. Yeung, P. Lin, P. Xu, S. Pophale, T. Sattasathuchana, V. Kale, W. Huhn, Y.H. He, Outcomes of OpenMP Hackathon: OpenMP application experiences with the offloading model (Part II), in: S. McIntosh-Smith, B.R. de Supinski, J. Klinkenberg (Eds.), OpenMP: Enabling Massive Node-Level Parallelism, Springer International Publishing, Cham, 2021, pp. 81–95.
[38] D.A. Davidson, Turbulence: An Introduction for Scientists and Engineers, Oxford University Press, Oxford, U.K, 2004.
[39] V. Eswaran, S.B. Pope, An examination of forcing in direct numerical simulations of turbulence, Comput. Fluids 16 (1988) 257–278.
[40] D.A. Donzis, P.K. Yeung, Resolution effects and scaling in numerical simulations of passive scalar mixing in turbulence, Physica D 239 (2010) 1278–1287.
[41] R.S. Rogallo, Numerical experiments in homogeneous turbulence, NASA Technical Memo 81315, NASA Ames Research Center, 1981.
[42] P.K. Yeung, K.R. Sreenivasan, S.B. Pope, Effects of finite spatial and temporal resolution on extreme events in direct numerical simulations of incompressible isotropic turbulence, Phys. Rev. Fluids 3 (2018) 064603.
[43] M. Lee, R. Ulerich, N. Malaya, R.D. Moser, Experiences from leadership computing in simulations of turbulent channel flows, IEEE Comput. Sci. Eng. 16 (2014) 24–31.
[44] K. Kandalla, K. McMahon, N. Ravi, T. White, L. Kaplan, M. Pagel, Designing the hpe cray message passing toolkit software stack for hpe cray ex supercomputers, in: Cray User Group 2023 Proceedings, 2023, https://cug.org/proceedings/cug2023_proceedings/at_a_glance.html.
[45] P.K. Yeung, Lagrangian investigations of turbulence, Annu. Rev. Fluid Mech. 34 (2002) 115–142.
[46] T. Gotoh, P.K. Yeung, Passive scalar transport in turbulence: a computational perspective, in: P.A. Davidson, Y. Kaneda, K.R. Sreenivasan (Eds.), Ten Chapters in Turbulence, Cambridge University Press, 2013.
[47] M.P. Clay, D. Buaria, P.K. Yeung, T. Gotoh, GPU acceleration of a petascale application for turbulent mixing at high Schmidt number using OpenMP 4.5, Comput. Phys. Commun. 228 (2018) 100–114.
[48] Y. Li, E. Perlman, M. Wan, Y. Yang, C. Meneveau, R. Burns, S. Chen, A. Szalay, G. Eyink, A public turbulence database cluster and applications to study Lagrangian evolution of velocity increments in turbulence, J. Turbul. 9 (2008) 1–29.
[49] H. Yao, P.K. Yeung, T.A. Zaki, C. Meneveau, Forward and inverse energy cascade in fluid turbulence adhere to Kolmogorov's refined similarity hypothesis, Phys. Rev. Lett. 132 (2024) 164001.
[50] Y. Tian, M. Woodward, M.G. Stepanov, C. Fryer, C. Hyett, D. Livescu, M. Chertkov, Lagrangian large eddy simulations via physics-informed machine learning, Proc. Natl. Acad. Sci. USA 120 (2023) 34.