Distributed-memory simulations of turbulent flows on modern GPU systems using an adaptive pencil decomposition library

Joshua Romero
NVIDIA Corporation
Santa Clara, CA, USA
joshr@nvidia.com

Pedro Costa
University of Iceland
Reykjavík, Iceland
pcosta@hi.is

Massimiliano Fatica
NVIDIA Corporation
Santa Clara, CA, USA
mfatica@nvidia.com

ABSTRACT

This paper presents a performance analysis of pencil domain decomposition methodologies for three-dimensional Computational Fluid Dynamics (CFD) codes for turbulence simulations, on several large GPU-accelerated clusters. The performance was assessed for the numerical solution of the Navier-Stokes equations in two codes which require the calculation of Fast-Fourier Transforms (FFT): a tri-periodic pseudo-spectral solver for isotropic turbulence, and a finite-difference solver for canonical turbulent flows, where the FFTs are used in its Poisson solver. Both codes use a newly developed transpose library that automatically determines the optimal domain decomposition and communication backend on each system. We compared the performance across systems with very different node topologies and available network bandwidth, to show how these characteristics impact decomposition selection for best performance. Additionally, we assessed the performance of several communication libraries available on these systems, such as OpenMPI, IBM Spectrum MPI, Cray MPI, the NVIDIA Collective Communication Library (NCCL), and NVSHMEM. Our results show that the optimal combination of communication backend and domain decomposition is highly system-dependent, and that the adaptive decomposition library is key in ensuring efficient resource usage with minimal user effort.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; • **Networks** → *Network performance evaluation*; • **Applied computing** → *Physical sciences and engineering*.

KEYWORDS

Parallel transpose, GPU accelerated systems, Computational Fluid Dynamics, Direct Numerical Simulation

ACM Reference Format:
Joshua Romero, Pedro Costa, and Massimiliano Fatica. 2022. Distributed-memory simulations of turbulent flows on modern GPU systems using an adaptive pencil decomposition library. In *Platform for Advanced Scientific Computing Conference (PASC '22), June 27–29, 2022, Basel, Switzerland*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3539781.3539797

PASC '22, June 27–29, 2022, Basel, Switzerland

ACM ISBN 978-1-4503-9410-9/22/06.
https://doi.org/10.1145/3539781.3539797

1 INTRODUCTION

Turbulence is the highly chaotic, three-dimensional and multi-scale state present in most large-scale fluid flows of practical interest. Direct Numerical Simulations (DNS) of the governing Navier-Stokes equations with high-fidelity in space and time necessary to capture the important small scale flow dynamics require very fine grids and long integration times. Supercomputers have been used for DNS since the seminal work of [15] in the early '70s and their growing performance and memory capacity are enabling the simulation of flows at larger Reynolds numbers (isotropic turbulence at $Re_\lambda = 2300$ with up to trillions of points [6], pipe flow at $Re_\tau = 6000$ [16], or channel flow at $Re_\tau = 8000$ [21]).

CFD is among the various computational physics domains where one-dimensional (slab) and two-dimensional (pencil) decompositions are commonly used to distribute 3D spatial data among parallel processes. For example, pseudo-spectral solvers often use such decompositions to compute three-dimensional FFTs [3]; direct, FFT-based finite-difference Poisson solvers rely on a combination of FFTs and Gauss elimination to achieve very fast, direct solutions [14, 17]; alternating-direction implicit methods (ADI) – commonly used as an approximate factorization of a Helmholtz equation for viscous diffusion terms – require the successive solution of one-dimensional tridiagonal systems along each direction [14].

In all these methods, the distributed slab or pencil data must be transposed, to collect complete input data that is local to each process. It is well known that the communication of data between processes during these transposes accounts for a large percentage of the computational cost. Investigations into optimizing 3D FFT and transpose performance on CPU systems have been performed [18], but GPU accelerated systems present unique challenges. For example, process local solver operations (e.g., FFTs or solving a sparse linear system), and packing operations, operating with fast GPU DRAM bandwidth, account for a smaller percentage of the wall time [1, 2, 22]. As such, it is even more imperative to select appropriate decomposition layouts to maximize network performance; however, making an optimal selection is far from trivial on modern GPU systems: there are complex multi-socket, multi-GPU node topologies with interactions between fast on-node peer-to-peer connections (e.g., NVIDIA NVLink) and lower bandwidth internode network connections.

Moreover, in addition to the hardware complexity, there are numerous libraries available for data communication between GPUs, both within the same node and across nodes of a cluster. Several MPI implementations can be found across the largest GPU systems available today, such as OpenMPI, IBM Spectrum MPI, and Cray MPICH, to name a few. Additionally, NVIDIA has introduced several GPU-specific communication libraries: the NVIDIA Collective

Communication Library (NCCL) [12] and NVSHMEM [13], a newer library implementing OpenSHMEM shared-memory routines on GPUs. With such a wide array of options, determining a priori the optimal pair of communication library and process decomposition to use for a given problem is nearly impossible, since it will depend on the specific software/hardware details of each machine.

In this work, we present a performance study of an adaptive pencil decomposition library across several large GPU clusters. This library is written with support for various communication backends, enabling runtime autotuning to determine the optimal process decomposition grid and communication backend that maximizes the transpose performance for the specific GPU system and computational setup. We applied this library to a pseudo-spectral solver for tri-periodic domains, and to an existing finite-difference solver for canonical turbulent flows. We assessed the performance of the library for the DNS of a decaying Taylor-Green vortex, and a pressure-driven turbulent channel. The results highlight the impact of the process decomposition layout and communication backend on the transpose – and, ultimately, the solver – performance, and how the runtime autotuning step can yield improved scaling over the conventional approach of implementing and using a single communication backend.

2 ADAPTIVE DECOMPOSITION LIBRARY

The adaptive pencil decomposition library discussed in this study was written in C++ and CUDA implementing a generic 2D pencil decomposition of 3D data with several distributed data transposition routines. The API is based on the popular 2DECOMP&FFT [8] library, and implements the same set of transpose operations. The library is capable of running any valid 2D decomposition of a given problem domain (which includes 1D slab decompositions), enabling comparative testing of different domain partitioning and its impact on overall performance. The library also contains Fortran bindings to enable a straightforward porting to HPC applications written in this language.

In addition to generally supporting all valid domain decomposition choices for a given grid, the library was written to support several communication backends (MPI Point-to-point, MPI All-to-all, NCCL, NVSHMEM, plus different staging strategies). This enables a combined runtime testing of both the grid decomposition layout and communication backend, to select the best-performing option during an autotuning process.

In the following sections, we proceed to discuss the decomposition library in detail, including the transposition implementations, available communication options, and autotuning procedure.

2.1 Pencil decomposition details

We consider 3D Cartesian grids with dimensions $X \times Y \times Z$, with the leading dimension in memory along X. The domain is decomposed across processes in an $R \times C$ grid, where the R, or row, processes are distributed along the Y dimension and C, or column, processes are distributed along the Z dimension. The communicator ranks are organized such that the rows correspond to contiguous sets of ranks. A labeled illustration of a sample decomposition is shown in Fig. 1.

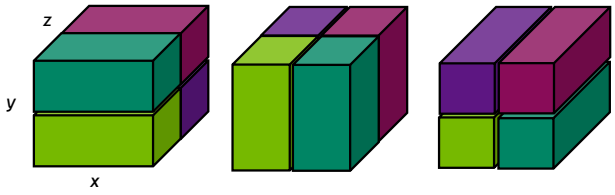

Figure 1: Illustration of a pencil decomposition of a domain into a 2×2 process grid. Each color represents data belonging to a different GPU. From left to right, the images show the domain distributed across X-axis aligned pencils, Y-axis aligned pencils, and Z-axis aligned pencils.

2.2 Transposition routines

We implemented a set of transpose routines necessary to perform 3D FFTs, following the routines available in the 2DECOMP library: `TranposeXtoY`, `TransposeYtoZ`, `TranposeZtoY`, and `TranposeYtoX`. Each routine implements the transposition from pencils aligned in one axis to another. For example, `TransposeXtoY` transposes X-axis aligned pencils to Y-axis aligned pencils. Each transpose operation is implemented using a sequence of a process local operation (e.g., a local data transposition or a packing operation), followed by a communication operation, specifically an all-to-all operation across either the row or column sub-communicator to swap pencil data among processes, followed by a second process local operation.

In addition to operating on pencil data buffers that preserve the original data memory layout, we implemented the option of using data buffers with the pencil axis as the leading dimension, which we will refer to as axis-contiguous pencils. While maintaining pencil buffers in the original memory layout can improve transpose performance by requiring fewer process local operations, FFT operations on these pencils may require strided memory accesses which introduce complexity to the performance analysis. Hence, for simplicity, we limit the testing to axis-contiguous pencils, where all transpositions require two local operations with a communication operation in between, so process local solver operations such as FFTs are conveniently applied to contiguous data arrays.

The breakdown of the sequence of operations used for each transpose routine is shown in Table 1. As each transpose routine performs similar operations, let us describe only one, say `TransposeXtoY`, in detail. The set of operations in `TransposeXtoY` transpose X-axis aligned pencils to Y-axis aligned pencils. The first operation is a local transpose of the local X-axis pencil data from an (X, Y, Z) memory layout to an (Y, Z, X) memory layout. This local transpose accomplishes two tasks. First, it transposes the pencil data to the target axis-contiguous Y-axis pencil memory layout (i.e., with Y as the leading dimension). Second, it splits the pencil data along the X-axis, as X is the trailing dimension of the transposed data. This results in a buffer suited for the all-to-all communication, where the data sent resides in contiguous blocks for each peer process. In our implementation, we use the `cutensorPermutation` operation from the cuTENSOR library [11] to perform these local transposes efficiently on the GPU.

The next operation is an all-to-all communication across *row* processes, as the Y-axis is split among these sets of processes. Note that

for transposes to and from the Z-axis pencils, the all-to-all communication is across *column* processes. The all-to-all communication will be discussed in more detail in the next section.

After the all-to-all operation, each process has the full set of Y-axis pencil data, with a sequence of data received from each peer process in contiguous chunks of memory. To reorganize the data into the desired output pencil memory layout, an unpacking operation is required, collating the data from the contiguous chunks to the appropriate locations in the Y-axis pencil buffer. For packing/unpacking operations in the library, we implemented a custom batched 3D memcopy kernel in CUDA.

2.3 Communication backends

There are numerous available libraries for multi-GPU communication, making it difficult to determine the best option for a given combination of computational problem, number of processes, and GPU system. For broad flexibility, the code implements the all-to-all communication for the transpose routines using several backends, enabling runtime selection of the one that provides the greatest performance across the available options. The communication libraries supported are MPI, the NVIDIA Collective Communication Library (NCCL), and NVSHMEM.

MPI is the de facto standard communication library in HPC, and was included due to its widespread use in the community. NCCL and NVSHMEM are two communication libraries provided by NVIDIA for fast GPU-to-GPU communication. NCCL is a communication library originally developed by NVIDIA for deep-learning training applications, with initial support for optimized GPU collectives, such as Allreduce and Broadcast. In recent releases, support for point-to-point (P2P) communication has been added via new `ncclSend` and `ncclRecv` operations, allowing wider usage of NCCL within applications outside the deep learning scope. NVSHMEM is a relatively new library released by NVIDIA to provide GPU-supported implementations of OpenSHMEM shared-memory communication routines, such as one-sided put and get operations.

For MPI, three options for the all-to-all data transfer between processes were developed. Two of the options implement the all-to-all using P2P communication routines (e.g., `MPI_Isend`, `MPI_Irecv`), and the other uses the existing `MPI_Alltoallv` (A2A) collective operation directly. Breaking down the two MPI P2P options, the first P2P implementation uses non-blocking send and receive APIs to schedule all the communication traffic to run concurrently without any *pipelining* of the local packing or transpose operations. In other words, the local operations before and after the all-to-all and the communication are completed sequentially, without any overlap. The second P2P implementation uses non-blocking receive and blocking sends to schedule communication between GPUs in pairs, pipelining the local operations before and after the all-to-all to overlap with communication. The `MPI_Send`/`MPI_Irecv` pairs are scheduled in either a butterfly/XOR pattern among ranks if the communicator size is a power of two; otherwise, buffers are exchanged in a ring pattern.

There are trade-offs between these two approaches. For the pipelined implementation, the cost of the local operations can be hidden behind the communication, but this requires splitting the local operations by peer process and launching GPU kernels per peer to perform the local transpose or (un)pack operation on the data sent and received from the peer processes. In strong scaling cases where the number of processes is large and the per peer buffer sizes are small, the latency of launching many small individual kernels can counteract any benefits from overlapping with communication. For the non-pipelined implementation, we can batch together many local operations into fewer kernel launches to reduce this latency; however, doing this limits the ability to overlap with communication, and the entire local operation runtime is exposed. We implemented both options and allow the autotuning process to decide when either option is better. Finally, for all MPI implementations, we use CUDA-aware MPI to operate directly on GPU memory buffers to avoid any extraneous memory movement from host to device for these communication routines.

For NCCL, a single all-to-all communication implementation was written, using `ncclSend` and `ncclRecv`, along with `ncclGroupStart` and `ncclGroupEnd` calls. With NCCL, the `ncclSend` and `ncclRecv` are blocking operations, with each call launching a single GPU kernel to execute a send or receive operation with a specified peer. To enable concurrent communication with multiple peers, the complete set of `ncclSend` and `ncclRecv` calls for the all-to-all operation is launched as a NCCL group call, established via launching a sequence of send and receive calls between calls to `ncclGroupStart` and `ncclGroupEnd`. This enables NCCL to execute a single kernel on each GPU to complete all required all-to-all communication concurrently, with a communication schedule determined by the NCCL backend, based on the topology detected during the communicator initialization.

For NVSHMEM, two all-to-all communication were written using a combination of non-blocking put (write) APIs. The two communication options closely match non-pipelined and pipelined MPI P2P options, replacing the MPI communication routines with equivalent NVSHMEM operations.

For the non-pipelined implementation, communication between P2P-accessible GPUs (e.g., GPUs connected via NVLink) is executed using host calls to `nvshmemx_putmem_nbi_on_stream`. These calls use copy engine resources on the GPUs via P2P memcopy operations, freeing up GPU compute resources (e.g, SMs) for other tasks. For peers not directly P2P accessible, the device function `nvshmemx_putmem_nbi` was called within a simple GPU kernel to launch the remaining put operations. While it is possible to use the host function `nvshmemx_putmem_nbi_on_stream` for every communication operation, this results in a single small GPU kernel launched to queue the communication operations for every non-P2P accessible peer, which suffers from latency issues when the number of peers in the communication grows large. It is therefore more efficient to use a simple kernel where multiple GPU threads schedule each required `nvshmemx_putmem_nbi`. For the pipelined implementation, we use the host `nvshmemx_putmem_nbi_on_stream` for all communication, as the communication is carried out per peer and negates any benefit from using a kernel to launch multiple NVSHMEM operations at once.

In all implementations, self copies were carried out using direct calls to device-to-device `cudaMemcpyAsync`. We observed that some CUDA-aware MPI implementations handle self copies with an unnecessary host-to-device data movement, so handling this

Table 1: Transpose routine operations breakdown.

Routine	Local Operation 1	Communication Operation	Local Operation 2
TransposeXtoY	Transpose (X, Y, Z) to (Y, Z, X)	All-to-all (row)	Unpack
TransposeYtoZ	Transpose (Y, Z, X) to (Z, X, Y)	All-to-all (column)	Unpack
TransposeZtoY	Pack	All-to-all (column)	Transpose (Z, X, Y) to (Y, Z, X)
TransposeYtoX	Pack	All-to-all (row)	Transpose (Y, Z, X) to (X, Y, Z)

case directly avoided any potential performance loss due to excess data movement.

2.4 Autotuning Procedure

The runtime autotuning of the process grid and communication backend is implemented as follows:

(1) In the library initialization call, the user provides target data global grid dimensions.
(2) During initialization, the library runs several timed trials of the full transposition sequence (e.g. X to Y, Y to Z, Z to Y and Y to X) for each valid process grid decomposition, and for each communication backend.
(3) The best combination of process grid decomposition and communication backend (lowest mean trial time) is chosen.

In addition to this full autotuning of the communication backend and process grid, the library also allows users to set the communication backend and autotune the process grid only, and vice-versa. Autotuning can also be fully disabled by fixing both settings.

As we will see, autotuning can provide significant performance benefits when running the same program across several computing clusters, since the optimal choice of backend and grid decomposition is not straightforward to predict.

3 LIBRARY USE CASES AND PERFORMANCE STUDY

To verify the correctness and real-world applicability of the adaptive pencil decomposition library, we applied it to the numerical simulation of two canonical turbulent flows: a decaying Taylor-Green vortex, and a pressure-driven turbulent channel. We used two different solvers: a pseudo-spectral one written from scratch in C++ and CUDA for the Taylor-Green case; and a second-order FFT-based finite difference solver, CaNS [5], based on CUDA Fortran for the channel case.

The library was applied to these codes, and we were able to test the resulting applications on some of the largest GPU systems in the world:

- Perlmutter at NERSC, an HPE system where each node has a single AMD Milan CPU, four 40GB A100 GPUs and two 100G Slingshot-10 NICs[1]
- Selene at NVIDIA, a system with DGX A100 nodes, each with a dual socket AMD Rome CPU, eight 80GB A100 GPUs and eight 200G Infiniband NICs.
- Marconi 100 at CINECA: an IBM system, where each node has a dual socket Power9 CPU, four 16GB V100 GPUs and one EDR 100G Infiniband NICs.

[1]It is important to note that Perlmutter's network will undergo significant upgrades in the near future that will substantially change the final size and network capabilities.

Since this study focuses on transpose performance in the context of distributed FFTs, a memory-bound and network-bandwidth-bound problem, a breakdown of available bandwidth, including the GPU DRAM bandwidth, intranode bandwidth between GPUs within a node, and internode bandwidth between GPUs across the network, is a useful reference to put performance numbers into context. Table 2 provides this information. The numbers in this table are provided per GPU, with the shared resources (e.g., NIC bandwidth) divided by the number of GPUs using the same connection. For systems with fully connected GPUs on the node via NVLink, we report only the NVLink bandwidth as the intersocket bus is not utilized.

Finally, we report the versions of the relevant software used in this study in Table 3. We note that a patched build of NVSHMEM was used on Perlmutter to overcome incompatibilities with the Cray PMI2 launcher, with no impact on the library performance.

Unless otherwise stated, we run with default MPI environment variables on each system as set by the system administration, under the assumption that these variables have been tuned for general usage on these clusters. For NCCL, we enable adaptive routing via the `NCCL_IB_SL` environment variable.

Table 2: System memory bandwidth resources per GPU.

System	GPU	Intranode	Internode
Selene	2 TB/s	300 GB/s (NVLink)	25 GB/s
Perlmutter (Phase 1)	1.5 TB/s	300 GB/s (NVLink)	6.25 GB/s
Marconi 100	0.9 TB/s	75 GB/s (NVLink) 32 GB/s (intersocket)	3.125 GB/s

Table 3: Software versions used in the performance experiments, for each system.

Library	Selene	Perlmutter	Marconi 100
CUDA	11.4.48	11.0.221	11.0.221
MPI	OpenMPI 4.0.5 UCX 1.11.0	Cray MPICH 8.1.11	Spectrum MPI 10.3.1.02rtm0
NCCL	2.11.4	2.11.4	2.11.4
NVSHMEM	2.4.1	2.4.1 (patched)	2.4.1
cuFFT	10.5.2.100	10.2.1.245	10.2.1.245
cuTENSOR	1.3.1	1.3.1	1.3.1

3.1 Application Details

CaNS – FFT-based finite difference solver. CaNS (Canonical Navier-Stokes) is a second-order finite-difference code for massively-parallel numerical simulation of canonical incompressible flows. The code uses a standard pressure correction method, with an FFT-based direct solver for the finite-difference Poisson equation. This Poisson solver is based on the method of eigenfunction expansions, and

is implemented to cover all combinations of boundary conditions valid for such a solver in a unified framework. In two directions, the grid is regular and the method employs fast Fourier/sine/cosine transforms to reduce the Poisson equation into a tridiagonal system at a low cost. This system is then solved along the third (Z) direction using Gauss elimination, which also allows for a non-uniform grid. Time is advanced with an explicit, three-step low storage Runge-Kutta scheme [20]. There are two versions of the code, one for CPU clusters[4] and another one for GPU accelerated clusters[5].

The GPU version of CaNS was modified to use the new communication library instead of the pre-existing GPU transpose routines based on MPI P2P, exposing all the new backends and auto-tuning capabilities. Here, the default pencil orientation outside the Poisson solver is along X (figure 1, left). The set of operations required to solve the Poisson equation requires four transposes per Poisson solve and is as follows: (1) perform one-dimensional FFTs along X; (2) transpose X-to-Y and perform FFTs along Y; (3) transpose Y-to-Z and solve the tridiagonal systems along Z with Gauss elimination; (4) and (5) perform the reciprocal operations, i.e., Z-to-Y and Y-to-X transposes, together with the inverse FFTs.

Pseudo-spectral solver. The Navier-Stokes solver used for the Taylor-Green vortex simulation is based on a pseudo-spectral Fourier-Galerkin method for the spatial discretization on a tri-periodic domain [3]. The equations are written in the rotational form using a velocity-vorticity formulation. Time advancement is performed with an explicit 4th-order Runge-Kutta method. Most operations are performed in Fourier space, with just the products associated with non-linear terms computed in physical space, and subsequently dealiased in Fourier space with a 2/3 rule. The non-linear term computation requires several FFT transforms, while all other computations are element-wise. The implementation follows the code presented in [9]. We should note that, unlike in CaNS, the solver has no additional communication costs arising from halo communication, since it does not perform stencil-type operations. Hence, the pseudo-spectral solver shows a more direct footprint of the transpose performance in the code.

3.2 CaNS Performance Study

Problem Description and Setup. We simulated turbulent channel flow, periodic along the streamwise (X) and spanwise (Y) directions, with no-slip/no-penetration boundary conditions at the top and bottom walls (along Z). We used a setup similar to that in [5], with doubled spanwise and tripled streamwise domain extents, which is closer to the dimensions of state-of-the-art turbulent channel flow DNS [7]. The flow is driven by a uniform pressure gradient that ensures a constant bulk velocity, on a domain with $N_x \times N_y \times N_z$ grid points, and channel dimensions $L_x \times L_y \times L_z = 18h \times 6h \times 2h$, where h is the channel half height. As per requirement of the solver, the grid is regular along the first two dimensions; along the Z direction, the grid is clustered at both walls as described in [5], with a grid-stretching parameter $a = 1.6$. We aimed at simulating a turbulent channel flow at friction Reynolds number $\mathrm{Re}_\tau \approx 590$. The Reynolds number based on the bulk velocity is set to $\mathrm{Re}_b = 12700$, estimated to match the target Re_τ. Additional details and verification of our implementation of this problem can be found in §A.1.

To investigate performance, we ran this channel flow problem on the Selene, Perlmutter, and Marconi 100 clusters, at scales from a single node (4 GPUs on Perlmutter/Marconi 100, 8 GPUs on Selene) up to 512 GPUs (128 nodes on Perlmutter/Marconi 100, 64 nodes on Selene). On each system, we ran problem sizes that approximately saturated the available GPU memory on a single node, and strong scaled the problem to 512 GPUs. The grid sizes to saturate the single node memory were different on each system due to differences in GPU models and node configurations and can be found in Table 4. We ran the largest grid in our study on Selene, as it has the largest single node GPU memory capacity with eight 80 GB A100 GPUs per node, while we ran the smallest grid on Marconi 100, as it has only four 16 GB V100 GPUs per node.

Table 4: Channel flow DNS grid dimensions by system.

System	$N_x \times N_y \times N_z$
Selene	$4068 \times 1536 \times 576$
Perlmutter	$2048 \times 768 \times 576$
Marconi 100	$2048 \times 640 \times 256$

At each scale tested, we ran our library in three different configurations. The first configuration fixes the communication backend to MPI P2P with pipelining, and manually sets the process grid to slabs of dimension $N_{\mathrm{GPU}} \times 1$. This configuration mimics a code that implements a fixed communication implementation for the transpose, which is very commonly MPI P2P based, and naively decomposes the problem into a 1D slab decomposition at all scales. The second configuration similarly fixes the communication backend to MPI P2P with pipelining, but autotunes the process grid. This configuration is used to measure the impact autotuning the process grid only can have on the performance results. Finally, the third configuration enables autotuning of both the communication backend and process grid to gauge any benefits of additionally tuning the communication backend selection in tandem with the process grid.

We should note that the version of CaNS used in this analysis has restrictions on the process decompositions allowed: N_x and N_y should be divisible by R; and N_y and N_z by C. As such, we limited the autotuning to these valid process decompositions only.

As our performance data was collected on clusters in production, we needed to consider potential network interactions with other jobs on the cluster, as well as variations in node allocations. To mitigate impacts of network variations due to the node allocations of our benchmarking jobs, the three configurations were run back-to-back within the same node allocation, with changes in node allocations only made when changing scale. While this does not account for potential network variation impacting data across the scales tested, it ensures a fair comparison between configurations and backend options using a consistent network topology between allocated nodes.

Performance Results and Discussion. The first set of figures (Figs. 2, 3, and 4), plots the wall-clock time per step of the channel flow test case from the single node configuration up to the 512 GPU scale for the three different library configurations. The marker types indicate the communication backend selected by the autotuner. The grid dimensions at the top of these figures present the process grids

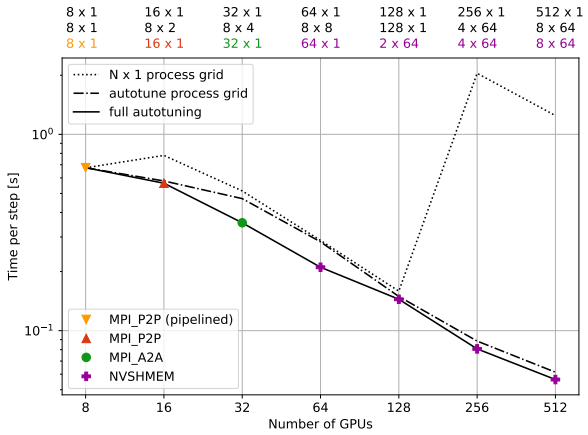

Figure 2: CaNS average wall-clock time per time step for channel test case with grid size $4068 \times 1536 \times 576$ on Selene. The different colored symbols denote the best-performing communication backend for the fully-autotuned run.

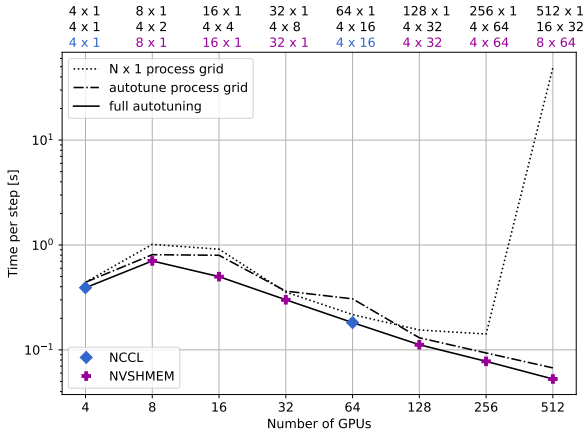

Figure 3: CaNS average wall-clock time per time step for channel test case with grid size $2048 \times 768 \times 576$ on Perlmutter.

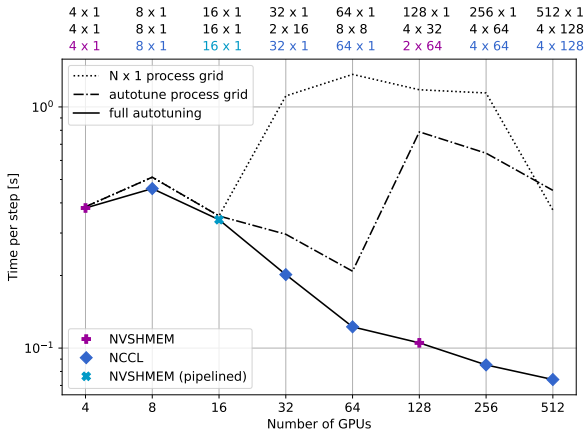

Figure 4: CaNS average wall-clock time per time step for channel test case with grid size $2048 \times 640 \times 256$ on Marconi 100.

selected by the autotuner for the three library configurations (top row corresponds to the slabs-only case, middle row corresponds to the autotuned grid only case, and bottom row corresponds to the fully autotuned case).

In general, we achieved the best performance with the fully autotuned configuration on all three clusters, at all scales tested. Comparing the fully autotuned results to the other configurations using the fixed MPI P2P with pipelining communication backend, we see different behavior across the systems. On Selene, we see that the slabs-only configuration is the worst-performing, with a large breakdown performance at the larger scales tested. Enabling grid autotuning allowed to recover this performance loss, but this was still ultimately limited in performance by the slower fixed communication backend at most of the scales tested. On Marconi 100, the other configurations perform even worse, with the fixed MPI P2P communication backend showing a breakdown in performance from medium to moderate scales. Autotuning the grid alone improves the performance at moderate scales, but is unable to find a competitive configuration to the fully autotuned configuration at the largest scales. In contrast, on Perlmutter, the other configurations maintain competitive performance with the fully autotuned configuration across the low to medium scales tested. At the largest scales, the slabs-only configuration performance drops and the grid autotuning alone is able to recover some performance that is close to, but behind the one achieved by the fully-autotuned configuration.

Another common feature in these results is a considerable performance slowdown that occurs between the first two data points in most of the configurations, which correspond to the transition from single to multi-node runs. This is most severe on Perlmutter, where the results do not recover single node performance until reaching 32 GPUs (8 nodes). On Marconi 100, the drop is less severe, with a recovery of single node performance at 16 GPUs (4 nodes). On Selene, the slabs-only configuration experiences this performance drop, while the other two configurations achieve speed up across this transition. These trends are a consequence of the multi-node communication, where the network links with significantly less available bandwidth compared to the fast on-node P2P like NVLINK are introduced into the transpose all-to-all communication. If this performance drop in the transpose is large enough to counter any local computation speedups from using more GPU resources, the overall performance drops. This drop is most severe on Perlmutter, as it currently has the largest ratio per GPU intranode bandwidth to internode bandwidth, with 48 times more intranode bandwidth to internode bandwidth (see Table 2). On the other hand, the two autotuned configurations on Selene are able to maintain a high enough transpose performance to overcome this performance loss; however, the scaling is not ideal, indicating that the transpose performance loss still overshadows some of the local compute performance scaling.

While these wall-clock time per step figures provide interesting insights, they do not illustrate the performance across the full range of communication backends and process grid decompositions. Figs. 5, 6 and 7 present the complete set of transpose timing data captured during autotuning for the different communication backends and grid decompositions on each system, for the setups in table 4. In these figures, the marker types are assigned to the

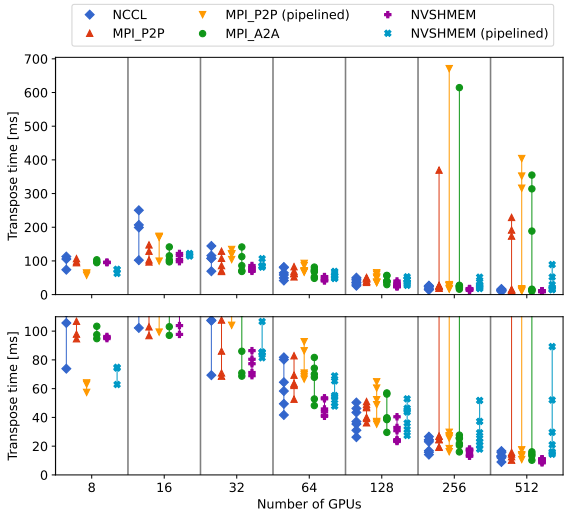

Figure 5: Average transpose trial times measured during autotuning on Selene, versus the number of GPUs. The top panel uses a larger ordinate range to pinpoint major differences in timings.

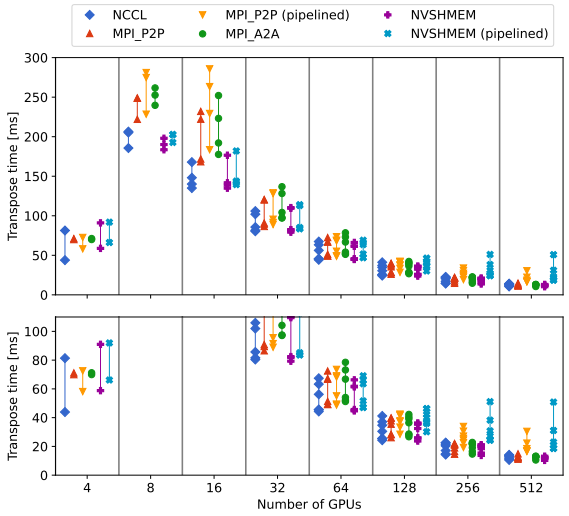

Figure 6: Same as in Fig. 5, but on Perlmutter (Phase 1).

different communication backends, with each point in these plots corresponding to a communication backend–grid decomposition pair. The markers are grouped by number of GPUs, and the vertical axis is the average transpose time measured by the library during autotuning.

In addition to these plots illustrating the overall picture, we include bar charts that more specifically break down the transpose performance across the tested grid decompositions at single node (Figs. 8, 9 and 10) and 32 node (Figs. 11, 12 and 13) scales.

On Selene, we see from Fig. 5 that there are differing ranges of timings by grid decomposition across communication backends, with NVSHMEM generally showing the least variability, while MPI

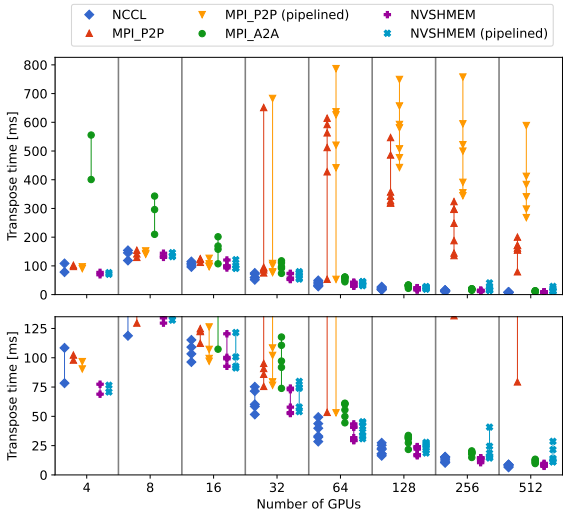

Figure 7: Same as in Fig. 5, but on Marconi 100.

backends vary the most at large scale. The pipelined implementations are the best at single node, but fall behind other backend options as the scale increases. At the smaller scales, the NCCL, non-pipelined MPI and NVSHMEM backends have comparable performance in the best case; NCCL and NVSHMEM perform the best at the larger scales. The single node timing breakdown in Fig. 8 shows the MPI P2P pipelined backend providing the greatest performance. Interestingly, while NCCL and NVSHMEM backends perform better with the slab decomposition (8×1), the non-pipelined MPI backends perform slightly better-using pencil decompositions. Moving onto the 32 node timing breakdown in Fig. 11, we see a drop in performance of the MPI backends and pipelined NVSHMEM backend for the slab configuration (256×1), while the slab configuration is among the fastest options for the other backends. Across all the backends, it can be seen that the best-performing decomposition is not using slabs, but a shallow pencil decomposition with either 4 or 8 row processes. This is an interesting result: Selene nodes contain 8 GPUs each, meaning these decompositions have a row communicator that is fully intranode. This indicates that, in some cases, a pencil decomposition with a fully intranode row communicator on a system with fast intranode P2P connections can outperform a slab decomposition, even though it requires all-to-all communication across both axes.

The transpose timing results for Perlmutter are plotted in Fig. 6. In a single node, the MPI P2P pipelined backend is among the best, but is outperformed by NCCL. At low to moderate scales, we see the NVSHMEM and NCCL backends outperform the MPI backends, while at large scale, the non-pipelined MPI backends provide similar performance to NCCL and NVSHMEM. Furthermore, the MPI backends at large scale show much less variability in performance by decomposition than those on Selene. The single node timing breakdown for Perlmutter in Fig. 9 shows that the NCCL backend in a slab (4×1) configuration outperforms the remainder by a significant margin, with the closest possible option (MPI P2P pipelined, 4×1) being around 30% slower. Looking at the 32 node timing breakdown in Fig. 12, the MPI implementations do not show a drop

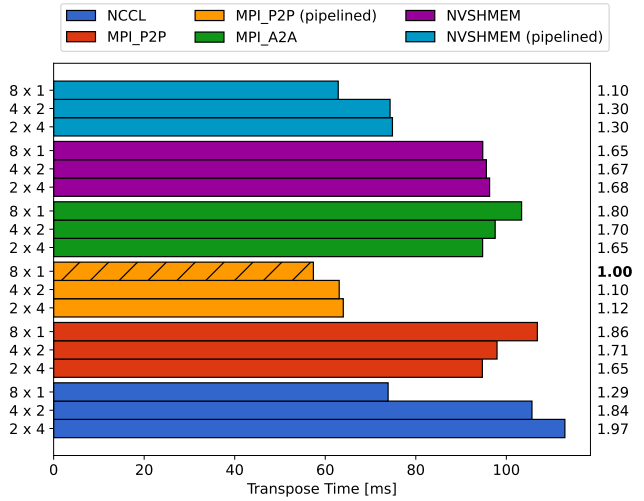

Figure 8: Single node (8 GPU) average transpose trial times measured during autotuning on Selene by backend and grid decomposition. In this and similar charts, the hatched bar indicates the configuration selected by the autotuner, and the values listed to the right of the plot are transpose times normalized by the selected case.

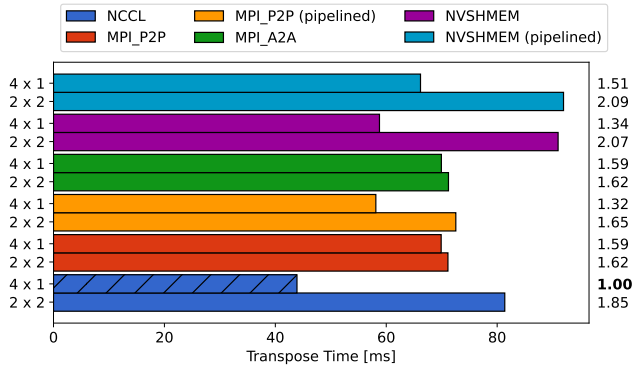

Figure 9: Single node (4 GPU) average transpose trial times measured during autotuning on Perlmutter (Phase 1) by backend and grid decomposition.

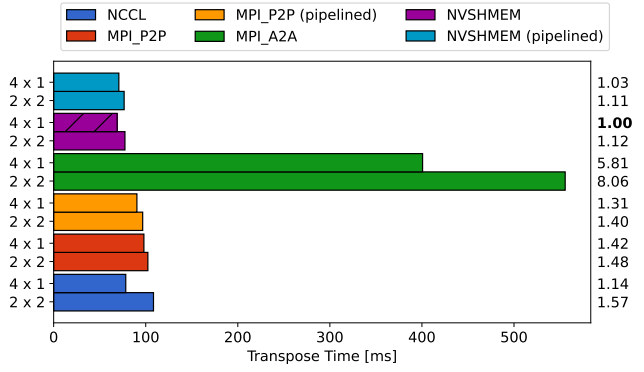

Figure 10: Single node (4 GPU) average transpose trial times measured during autotuning on Marconi by backend and grid decomposition.

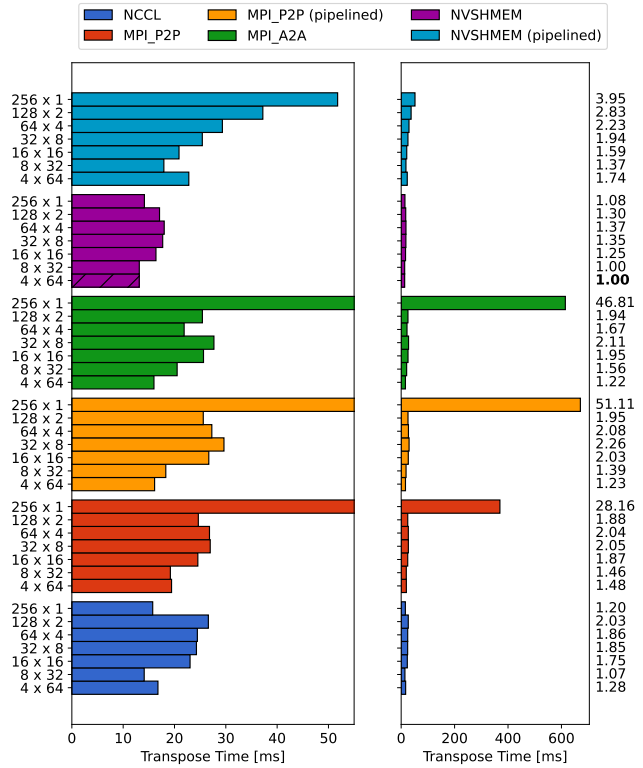

Figure 11: 32 node (256 GPU) average transpose trial times measured during autotuning on Selene by backend and grid decomposition.

in performance in slab configurations in contrast to the results on Selene. For the NCCL and non-pipelined MPI backends, the slab configuration (128×1) is among the best-performing options, but similar to Selene, one of the shallow pencil decompositions with a fully intranode row communicator performs slightly better for each backend.

Finally, for Marconi 100, we see from Fig. 7 that while the NCCL and NVSHMEM backends show similar timing ranges by grid decomposition for each backend, the MPI backends have more varied behavior, with the MPI P2P backends showing wide variability at large scale. Interestingly, on this system the MPI A2A backend is one of the worst-performing backends at small scales, but outperforms the other MPI P2P backend options at larger scales and reaches comparable performance to NVSHMEM and NCCL. On this system, the NCCL backend achieves the best performance at the majority of the tested scales. Of note at large scales on this system, the MPI P2P backends do not achieve comparable performance to the others, on any of the grid configurations tested. The single node timing breakdown for Marconi 100 in Fig. 10 further illustrates the poor performance of the MPI A2A backend relative to the other options and that all the options show a slight performance advantage for the slab (4×1) configuration. Focusing on the 32 node performance chart in Fig. 13, we observe the reduction in performance of the MPI P2P backends across all the tested grid configurations, with the MPI A2A achieving better performance. For the NVSHMEM and pipelined MPI P2P backend options, we

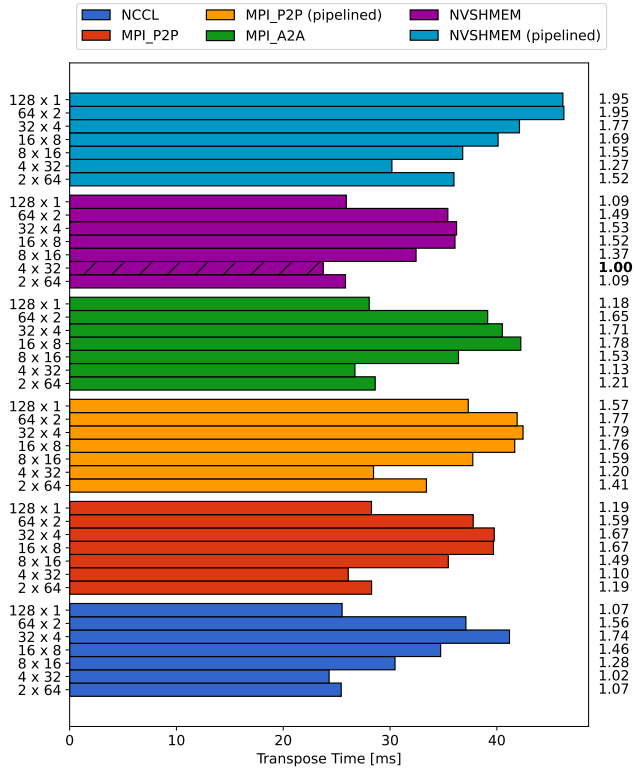

Figure 12: 32 node (128 GPU) average transpose trial times measured during autotuning on Perlmutter (Phase 1) by backend and grid decomposition.

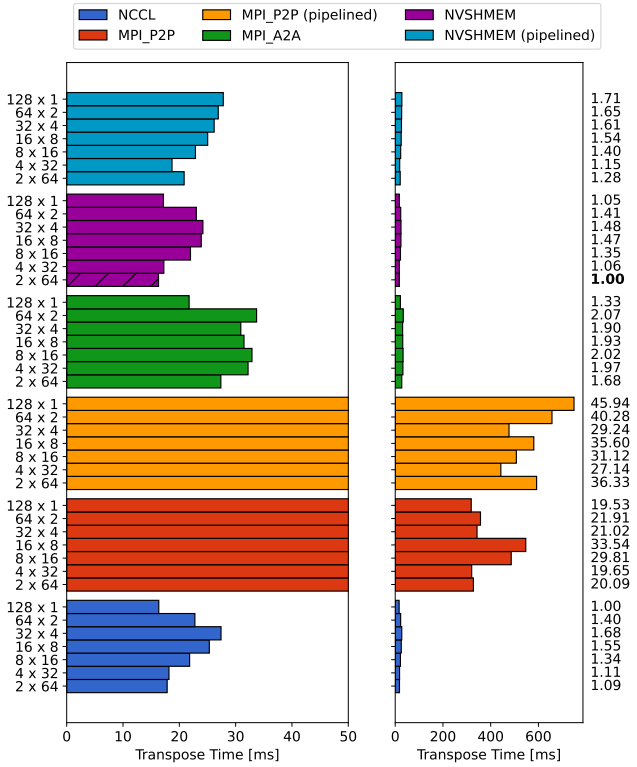

Figure 13: 32 node (128 GPU) average transpose trial times measured during autotuning on Marconi 100 by backend and grid decomposition.

see a performance improvement using pencil decompositions with an intranode row communicator but this trend does not hold for the NCCL or other MPI backend options, where the slab (128×1) shows the best performance.

Considering all these data as a whole, we can make a few general observations. First, there is no single dominant communication backend across all the tested systems and scales. While the NCCL and NVSHMEM options show the best large scale performance on Selene and Marconi 100, it is the MPI and NVSHMEM backends that show the best large scale performance on Perlmutter. Considering single node and smaller scales tested, NCCL and NVSHMEM remain the best on Marconi 100, while on Selene and Perlmutter, NVSHMEM falls behind in performance relative to the NCCL or MPI backend options. A second observation is that even when using a fixed communication backend, selecting an appropriate grid decomposition can yield performance increases over slabs, but the magnitude of this performance increase (if any), can vary across backends, even at equivalent system scales.

In summary, selecting the highest-performing communication backend and grid decomposition for a given problem and system scale is not easily captured into a well-defined set of principles, and motivates the need for autotuning procedures like that implemented in the current work. We see that, for this channel flow application, the autotuning procedure developed can successfully identify and run the best case configuration for the problems tested.

3.3 Pseudo-spectral Performance Study

Problem Description and Setup. Using the pseudo-spectral solver, we simulated the Taylor-Green case, a common benchmark in several papers and workshops [4, 9]. We ran the simulation matching the physical parameters of the simulation in [19], for a periodic box of dimension $[0, 2\pi]^3$, with an initial two-dimensional Taylor-Green flow with a Reynolds number set at 1600, using unit reference length and velocity scales. Additional details and verification of our implementation of this problem can be found in §A.2.

In contrast to the performance study in §3.2, we fixed the grid resolution to 1024^3 for all tests. As such, we restricted this study to the Selene and Perlmutter clusters only, due to the availability of similar GPU models available on each system. On these systems, we ran this problem at scales from 4 GPUs up to 512 GPUs, measuring the wall-time to complete 1000 steps of the simulation. For each scale, we configured the library to use three different fixed communication backends (NCCL, MPI P2P, and MPI A2A) and enabled autotuning of the process grid configuration only.

Performance Results and Discussion. Fig. 14 presents the performance results and scaling obtained on Selene and Perlmutter by communication backend used. Similar to the figures in the previous section, the process grid dimensions selected by the autotuner are presented above the plot for each case, with the colors indicating the communication backend used. The top three rows are Selene, while the bottom three rows are the selected grids on Perlmutter.

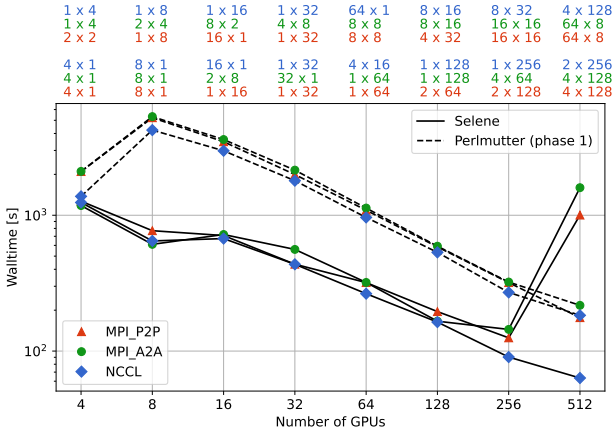

Figure 14: Taylor-Green simulation wall times on a 1024^3 grid, running 1000 time steps. The different colored symbols denote the communication backend used.

Recall that Selene has eight 80GB A100 GPUs per node and eight NICs, while Perlmutter has four 40GB A100 per node with two NICs. The GPUs on a node on both systems are fully connected via NVLink, but the 80GB version has more local memory bandwidth ($\approx$ 2 TB/s) than the 40GB version ($\approx$ 1.5 TB/s) due to the use of HBM2e memory with higher memory clock. Both the FFT transforms and the local transposes performed by the library, being memory-bound workloads, will run faster on the 80GB version. This explains the small gap in the fastest runtimes achieved on 4 GPUs between Selene and Perlmutter. On Perlmutter at this scale, NCCL is shown to outperform the MPI backends which is consistent with the results for CaNS.

At the transition from intra- to inter-node processing (between 4 to 8 GPUs on Perlmutter, and 8 to 16 GPUs on Selene), we observe degradation in scaling due to the introduction of slower network communication to the simulation. As expected, due to the much larger gulf between intra-node and inter-node bandwidth on Perlmutter (phase 1), the degradation is more severe and does not recover until 64 GPUs are used. In contrast, Selene recovers from this performance drop at a 32 GPU scale.

From the moderate to larger scales tested, we see that the scaling trend on both Perlmutter and Selene is very similar. This suggests that the expected improvement in network performance on the completed Perlmutter system, which should reduce the performance loss at the intra- to inter-node transition, may result in performance parity between the two systems. Finally, on Selene we see a performance drop in the MPI backends at the largest scale tested, similar to what was observed with CaNS, while on Perlmutter the MPI A2A backend is the most performant for this problem at this scale.

4 CONCLUSIONS

While this work has focused on several specific clusters with NVIDIA GPUs, the trend towards having fast multi-GPU nodes with increasingly high bandwidth interconnects and relatively lower network bandwidth is common across all major vendors. With this in mind, codes that rely on communication-intensive algorithms, such as spectral CFD solvers or many other applications that rely on parallel multi-dimensional FFTs, may be penalized by the performance implications of this trend. As seen in the results of this performance study, careful selection of communication libraries and process decomposition for the system used is essential in achieving good performance, and can result in substantial savings in compute time. The decomposition library used in this study, with autotuning functionality to sweep through different communication backends and process decompositions, has shown that this type of approach can be useful for application developers to determine optimal configurations for their programs.

The decomposition library discussed in this study will be available at https://github.com/NVIDIA/cudecomp.

ACKNOWLEDGMENTS

This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 using NERSC award NERSC DDR-ERCAP0022341. We also acknowledge the use of the Marconi 100 supercomputer hosted by CINECA in Italy. We would like to thank Gregory Ruetsch for his contribution to the Fortran bindings of the library.

REFERENCES

[1] Alan Ayala, Stanimire Tomov, Azzam Haidar, and Jack Dongarra. 2020. heFFTe: Highly efficient FFT for exascale. In *International Conference on Computational Science*. Springer, Springer, 262–275.
[2] Alan Ayala, Stanimire Tomov, Piotr Luszczek, Sébastien Cayrols, Gerald Ragghianti, and Jack Dongarra. 2021. *Interim Report on Benchmarking FFT Libraries on High Performance Systems*. Technical Report.
[3] Claudio Canuto, M Yousuff Hussaini, Alfio Quarteroni, A Thomas Jr, et al. 2012. *Spectral methods in fluid dynamics*. Springer Science & Business Media.
[4] Pedro Costa. 2018. A FFT-based finite-difference solver for massively-parallel direct numerical simulations of turbulent flows. *Comput. Math. with Appl.* 76, 8 (2018), 1853–1862.
[5] Pedro Costa, Everett Phillips, Luca Brandt, and Massimiliano Fatica. 2021. GPU acceleration of CaNS for massively-parallel direct numerical simulations of canonical fluid flows. *Comput. Math. with Appl.* 81 (2021), 502–511.
[6] Takashi Ishihara, Koji Morishita, Mitsuo Yokokawa, Atsuya Uno, and Yukio Kaneda. 2016. Energy spectrum in high-resolution direct numerical simulations of turbulence. *Phys. Rev. Fluids* 1, 8 (2016), 082403.
[7] Myoungkyu Lee and Robert D Moser. 2015. Direct numerical simulation of turbulent channel flow up to. *J. Fluid Mech.* 774 (2015), 395–415.
[8] Ning Li and Sylvain Laizet. 2010. 2DECOMP & FFT – a highly scalable 2D decomposition library and FFT interface. In *Cray User Group 2010 Conference*. 1–13.
[9] Mikael Mortensen and Hans Petter Langtangen. 2016. High performance Python for direct numerical simulations of turbulent flows. *Comput. Phys. Commun.* 203 (2016), 53–65.
[10] Robert D Moser, John Kim, and Nagi N Mansour. 1999. Direct numerical simulation of turbulent channel flow up to Re τ= 590. *Phys. Fluids* 11, 4 (1999), 943–945.
[11] NVIDIA Developer 2021. cuTENSOR. https://developer.nvidia.com/cutensor
[12] NVIDIA Developer 2021. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl
[13] NVIDIA Developer 2021. NVSHMEM: NVIDIA. https://developer.nvidia.com/nvshmem
[14] Paolo Orlandi. 2000. *Fluid flow phenomena: a numerical toolkit*. Vol. 55. Springer Science & Business Media.
[15] Steven A. Orszag and G. S. Patterson. 1972. Numerical Simulation of Three-Dimensional Homogeneous Isotropic Turbulence. *Phys. Rev. Lett.* 28 (Jan 1972), 76–79. Issue 2.
[16] Sergio Pirozzoli, Joshua Romero, Massimiliano Fatica, Roberto Verzicco, and Paolo Orlandi. 2021. One-point statistics for turbulent pipe flow up to $Re_\tau \approx 6000$. *J. Fluid Mech.* 926 (2021).
[17] Ulrich Schumann and Roland A Sweet. 1988. Fast Fourier transforms for direct solution of Poisson's equation with staggered boundary conditions. *J. Comput.*

Phys. 75, 1 (1988), 123–137.
[18] Sukhyun Song and Jeffrey K. Hollingsworth. 2016. Computation–communication overlap and parameter auto-tuning for scalable parallel 3-D FFT. *J. Comput. Sci.* 14 (2016), 38–50. https://doi.org/10.1016/j.jocs.2015.12.001 The Route to Exascale: Novel Mathematical Methods, Scalable Algorithms and Computational Science Skills.
[19] Wim M Van Rees, Anthony Leonard, Dale I Pullin, and Petros Koumoutsakos. 2011. A comparison of vortex and pseudo-spectral methods for the simulation of periodic vortical flows at high Reynolds numbers. *J. Comput. Phys.* 230, 8 (2011), 2794–2805.
[20] Alan A Wray. 1986. *Very low storage time-advancement schemes.* Technical Report. Internal Report, Moffett Field, CA, NASA-Ames Research Center.
[21] Yoshinobu Yamamoto and Yoshiyuki Tsuji. 2018. Numerical evidence of logarithmic regions in channel flow at $Re_\tau = 8000$. *Phys. Rev. Fluids* 3, 1 (2018), 012602.
[22] Xiaojue Zhu, Everett Phillips, Vamsi Spandan, John Donners, Gregory Ruetsch, Joshua Romero, Rodolfo Ostilla-Mónico, Yantao Yang, Detlef Lohse, Roberto Verzicco, Massimiliano Fatica, and Richard J.A.M. Stevens. 2018. AFiD-GPU: A versatile Navier–Stokes solver for wall-bounded turbulent flows on GPU clusters. *Comput. Phys. Commun.* 229 (2018), 199–210.

A VERIFICATION AND VALIDATION

A.1 CaNS

The first verification of the correctness of the simulations was to confirm that the corrected velocity field in this direct solver is, to machine precision, divergence-free. Subsequently, we validated our method for a wall-bounded turbulence setup by comparing against known results from results of Ref. [10] and with the smaller domain setup simulated in CaNS-GPU [5]. We ran our verification case on Selene, using a grid size of $4068 \times 1536 \times 576$, on 256 A100 GPUs. The simulation ran for 200 000 time steps (about 400 bulk flow time units).

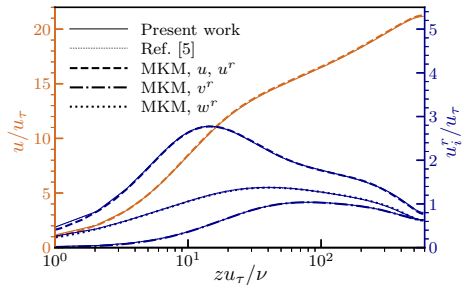

Figure 15: Profiles of mean streamwise velocity and root mean square velocities, in inner-scaling, i.e., scaled by the wall-friction velocity u_τ, versus the inner-scaled wall distance zu_τ/ν, with ν the fluid kinematic viscosity. The data is compared to that of Ref. [5] for the original CaNS-GPU implementation at the same Reynolds number but on a smaller domain, and for the seminal $\mathrm{Re}_\tau = 590$ case of [10] (MKM).

The results show excellent agreement with the dataset of Ref. [10], and of the simulations with a smaller domain in [5]. The mean friction Reynolds number obtained was $\mathrm{Re}_\tau = 584.45$, with a difference of less than 0.2% of the smaller domain results in [5]. Moreover, the inner-scaled mean velocity profile, Reynolds stresses also show excellent agreement with the data. We show in figure 15 the profiles of the mean stream-wise velocity and the root mean square of the three velocity components, compared to the Refs. [5, 10], to illustrate the agreement.

A.2 Pseudo-spectral solver

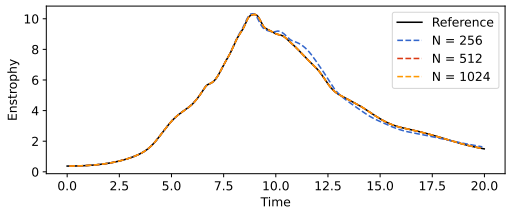

Figure 16: Taylor Green results: time evolution of enstrophy for on $256^3, 512^3$ and 1024^3 grids, compared to the reference data from [19].

For the Taylor Green case, we compare the time evolution of the enstrophy predicted by our pseudo-spectral solver to those of [19] at several grid resolutions. Figure 16 shows the great agreement with the reference data from [19], once the resolution is sufficient.