Sustained Petascale Direct Numerical Simulation on Cray XC40 Systems (Trinity, Shaheen2 and Cori)

Bilel Hadri
King Abdullah University of Science and Technology
KAUST Supercomputing Lab
Thuwal, Saudi Arabia
bilel.hadri@kaust.edu.sa

Matteo Parsani
King Abdullah University of Science and Technology
Extreme Computing Research Center
Thuwal, Saudi Arabia
matteo.parsani@kaust.edu.sa

Maxwell Hutchinson
Citrine Informatics
Redwood City, California, USA
maxhutch@gmail.com

Alexander Heinecke
Intel Corporation
Santa Clara, California, USA
alexander.heinecke@intel.com

Lisandro Dalcin
King Abdullah University of Science and Technology
Extreme Computing Research Center
Thuwal, Saudi Arabia
lisandro.dalcin@kaust.edu.sa

David Keyes
King Abdullah University of Science and Technology
Extreme Computing Research Center
Thuwal, Saudi Arabia
david.keyes@kaust.edu.sa

***Abstract*—**
We present in this paper a comprehensive performance study of highly efficient extreme scale direct numerical simulations of secondary flows, using an optimized version of NeK5000. Our investigations are conducted on various Cray XC40 systems, using the very high-order spectral element method. Single-node efficiency is achieved by auto-generated assembly implementations of small matrix multiplies and key vector-vector operations, streaming lossless I/O compression, aggressive loop merging and selective single precision evaluations. Comparative studies across different Cray XC40 systems at scale, Trinity (LANL), Cori(NERSC) and ShaheenII(KAUST), show that a Cray programming environment, network configuration, parallel file system and Burst buffer all have a major impact on the performance. All the 3 systems possess a similar hardware with similar CPU nodes and parallel file system, but they have a different network theoretical bandwidth, a different OS and different CDT versions of the programming environment. Our study reveals how these slight configuration differences can be critical in terms of performance of the application. We also find that using 294,912 cores (9216 nodes) on Trinity XC40 sustaines the petascale performance, and as well 50% of peak memory bandwidth over the entire solver (500 TB/s in aggregate). On 3072 KNL nodes of Cori, we reach 378 TFLOP/s with an aggregated bandwidth of 310 TB/s, corresponding to time-to-solution 2.11× faster than obtained with the same number of Haswell nodes.

***Keywords*-XC40, Haswell, KNL, Nek5000, Scaling Performance, Regression, Cray Programming Environment**

I. Introduction

The turbulent flow in a square duct represents a canonical case of wall-bounded turbulence, that differs from the well-studied cases of turbulent channels and pipes in the appearance of secondary flows. Secondary flows are characterized by a low amplitude fluid velocity in the cross-sectional plane, *i.e.* orthogonal to the primary flow direction, that contributes to increased mixing. Typically, secondary structures become apparent in the time-averaged flow, where they take the form of corner vortices.

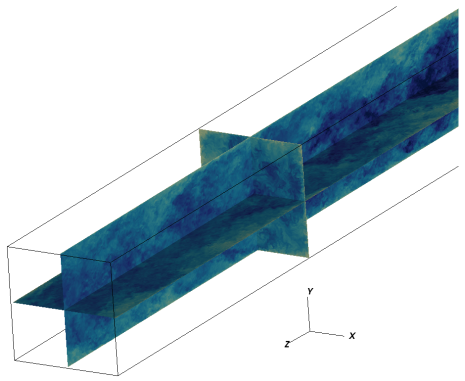

Figure 1. Three slices of the magnitude of the velocity in developing square duct flow at bulk Reynolds number of $Re = 100,000$.

Most wall-bounded internal and external flows found in technology and nature develop secondary flows of various strengths. Important application areas are turbo-machinery and heat exchangers, where rectangular ducts and diffusers are ubiquitous. Secondary flows are encountered even in external flows, *e.g.* flows between the wing and fuselage of an airplane. However, the origin and nature of these secondary corner vortices are still open questions; neither a complete theory nor a sufficiently predictive computational tool are available for such cases. A substantially improved understanding of their generation mechanisms would undoubtedly pave the way for technological improvements in real-world applications. For instance, an accurate description of secondary flows would allow the prediction of several integral and fluctuating quantities, such as the head loss (pumping power) [1] and the wall heat flux. Secondary flow can also crucially affect particle transport, as recently shown by [2]; the spatial homogeneity of the particle phase is completely changed by even moderate amplitude secondary flows, with implications for the process industry.

Here, we aim at simulating a turbulent flow in a duct, at a sufficiently high Reynolds number to be of practical interest, comparable to the highest Reynolds numbers studied in channels and pipes by [3]. For all runs, the bulk Reynolds number, based on the duct cross-section, is $Re = 100,000$, corresponding to a friction Reynolds number $Re_\tau > 2000$, and passive scalar mixing is included with Prandtl number $Pr = 1$. The runs differ in their stream-wise extent from $12h$ to $48h$, where h is the duct half-width, with the largest case having 69 billion grid points and 5 variables per point (pressure, three velocity components, and a passive scalar which corresponds to the temperature).

The next generation of flow solvers will undoubtedly be large eddy simulations (LES) and direct numerical simulations (DNS) based on numerical discretisations that combine high-order accuracy with unstructured grids as pointed out in [4]. The incompressible Navier–Stokes solver (NEKBOX) available by [5], a version of Nek5000 (c.f. [6]) optimized for tensor product geometries, is a forward-looking example of such a solver where high-order accurate schemes are co-designed for current petascale computing architectures. Nek5000 is an open-source code that implements the spectral element method (SEM) with tunable order for solving the three-dimensional (3D) incompressible Navier–Stokes equation on an unstructured mesh of tensor product elements for a wide range of scientific applications. Those include fluid flow, thermal convection, combustion, magnetohydrodynamics, and electromagnetics. Through a collaboration with hardware vendors and software technologies projects, Nek5000 has been selected by the Center for Efficient Exascale Discretizations (CEED), a co-design center within the U.S. Department of Energy (DOE) Exascale Computing Project (ECP), to help applications leverage future exascale systems by exploiting the hardware efficiently and effectively, by delivering a significant performance gain over conventional low-order methods.

Previous work on duct flows at low Reynolds number (see *e.g.* the review in [7]) identified secondary flows of the second kind, *i.e.* corner vortices due to cross stream gradients of the turbulent stresses. These secondary flows are easily visible as a deformation of the mean velocity contours. However, a quantification as a function of the Reynolds number, wall roughness, the span-wise aspect ratio of the duct, *etc.* is not yet available. Nonetheless, RANS models have been developed through appropriate anisotropic modeling of the Reynolds stresses [8] although the predictability of these models is not overarching.

Partial differential equation (PDE) solvers have been featured in many recent Gordon Bell competitions. The compressible Euler equations solved in [9] are hyperbolic equations on a uniform grid; they have no global communication other than a scalar reduction to synchronize the time step. Similarly, the elastic wave equations solved in [10] are hyperbolic, with a global synchronization only of the time-step and only at initialization. The performance of these types of codes is achieved by efficient on-node code, and facilated by easily identified high arithmetic intensity kernels, and asynchronous nearest-neighbor communication.

It is a widely held belief that multi-tiered parallelism, *i.e.* MPI+X, and asynchronous communication are required to scale on modern parallel architectures. In contrast, Nek5000 and NEKBOX are pure MPI with synchronous communication interfaces. It is worth noting that this execution model leads even on many-core processors (Intel Xeon Phi) to excellent performance results.

Nek5000, the basis of NEKBOX, received a Gordon Bell award in 1999 through the work of [11], and has been used extensively at scale. Previous Nek5000 simulations have gone as high as 8 billion gridpoints and spectral order 21. In Section IV, we present performance results up to 69 billion gridpoints and spectral orders of 15 and 31.

Performing a direct numerical simulation becomes computationally prohibitive with increasing Reynolds number ($\sim Re^{9/4}$, [12]), yet the results at a high Reynolds number regime, are the most interesting ones. To attain such Reynolds numbers, it is essential to design a code that is performant, highly parallelizable and robust. To this end, this study exploits the low computational cost of tensor product representation of a spatial discretization [13], the highly parallelizable additive Schwarz domain decomposition method presented in [14], the low communication cost of spectral element methods analyzed in [13], and the reusability of the solution space, via projections introduced in [15]. The parallelism reduces the problem to small matrix-matrix multiplications, c.f. [16], which are at the core of the entire solver and constitute the biggest part of the computational work per process.

II. Contributions, Algorithm and Implementation

A. Summary of contributions

Performance scale study on different XC40 systems. The performance at scale used three different Cray XC40 systems, ShaheenII (KAUST), Cori (NERSC) and Trinity(LANL). These systems exhibit similar hardwares with similar CPU nodes and parallel file system but with different networking between the groups, different operating systems, and different CDT versions of the programming environment, which can be critical in terms of applications' performance. The Cray Developer Toolkit (CDT) module effectively changes the default version of a software component modulefile to be the version associated with that specific CDT release. Using more recent packages may result in faster execution of the code.

Auto-generated assembly implementations for small matrix multiplies. Very high order, 15 and 31, spectral elements result in arithmetically intensive kernels near compute-bandwidth parity. To ensure optimal performance on a variety of instruction sets, we use an enhanced LIBXSMM presented in [17] to auto-generate the assembly implementations with the largest available vector instructions. Enhancements target NEKBOX tensor product operations and NEKBOX reproducers are part of LIBXSMM's regression tests. LIBXSMM shortens the execution time of the simulation without increasing the power consumption of the supercomputer. In particular, we observe a power saving of approximately 30% compared to the execution based on the compiled native FORTRAN on common Intel Xeon platforms.

Aggressive inter-procedural loop merging. To maximize cache utilization, loops over elements are aggressively coalesced. When data are written but not re-used, non-temporal stores avoid read-for-ownership memory operations. Nearest-neighbor communication interfaces are implemented element-wise, improving the locality of the buffer loading and providing opportunities for communication and computation overlap. The memory load of the solvers is reduced by a factor of up to 3.

Reduction in global coarse problem size. Very high order spectral elements with hybrid Schwarz multigrid preconditioning significantly reduce the size of the global coarse problem, improving scalability in the elliptic solve. Compared to the common case of spectral order $p = 7$, $p = 15$ and $p = 31$ reduce the problem size by $8\times$ and $64\times$, respectively.

Selective evaluation in single precision. Single-precision preconditioning is implemented to reduce costs when double-precision is unnecessary. This feature has an insignificant effect on the number of iterations and memory bandwidth rate, resulting in a speedup of $2\times$.

I/O with lossless compression. An already strong I/O performance is supplemented by lossless LZ4 compression. The compression is sufficiently light to hide behind the disk's I/O bottleneck.

Generalizable achievements. Because SEM shares many properties with compact high order continuous and discontinuous methods for incompressible and compressible Navier–Stokes equations on unstructured grids, nearly all the innovations reported herein can readily benefit a wide community of scientists. The new algorithms and optimizations live entirely in native code and libraries, making performance portable across homogeneous for today's and future architectures.

B. Overview of algorithm

NEKBOX solves the incompressible Navier–Stokes equations, given here in non-dimensional form by

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u} = -\nabla p + \frac{1}{Re}\nabla^2\mathbf{u} + \mathbf{f}, \qquad \nabla\cdot\mathbf{u} = 0, \quad (1)$$

where $\mathbf{u}$, p, Re and $\mathbf{f}$ are the velocity, the pressure, the Reynolds number, and the volume force, respectively.

Time integration is performed with an explicit extrapolation (EX) of nonlinear and forcing terms nested within the implicit backwards difference formula (BDF) for the Stokes part of the problem. In semi-discrete form, (1) becomes:

$$\sum_{j=0}^{k} \frac{b_j}{\Delta t}\mathbf{u}^{n-j} = - \nabla p^n + \frac{1}{Re}\nabla^2\mathbf{u}^n + \underbrace{\sum_{j=1}^{k} a_j [N(\mathbf{u}^{n-j}) + \mathbf{f}^n]}_{\mathbf{F}_n(\mathbf{u},\mathbf{f})} \quad (2)$$

where we denote the nonlinear operator $\mathbf{u}\cdot\nabla\mathbf{u} = N(\mathbf{u})$ and b_k, a_k are the coefficients of the BDF and EX schemes, respectively. Herein, these coefficients are chosen to achieve third-order accuracy (BDF3/EX3, as described in [6]).

Taking the divergence results in the pressure Poisson equation:

$$\nabla^2 p^n = \nabla\cdot\mathbf{F}_n(\mathbf{u},\mathbf{f}), \quad (3)$$

which is decoupled from the velocity at t^n via the explicit extrapolation, inserting a splitting error of the same order as the time integration. Finally, the gradient of the pressure is added to the right-hand side (RHS) of the velocity Helmholtz equation:

$$\frac{1}{Re}\nabla^2\mathbf{u}^n - \frac{b_0}{\Delta t}\mathbf{u}^n = \nabla p^n + \mathbf{F}_n(\mathbf{u},\mathbf{f}) + \sum_{j=1}^{k}\frac{b_j}{\Delta t}\mathbf{u}^{n-j}. \quad (4)$$

An advection-diffusion equation for passive scalars is solved via a similar Helmholtz equation.

From here, the algorithm is straightforward: 1) construct the RHS of the pressure Poisson equation; 2) solve the pressure Poisson problem; 3) construct the RHS for the velocity and scalar Helmholtz equations; 4) solve the four Helmholtz problems. In this sense, NEKBOX is primarily concerned with solving Poisson and Helmholtz problems.

Before entering the linear solvers, the RHS is projected into the space of the most recent solutions, typically between 8 and 32 solutions for the Poisson equation and 2 to 8 solutions for the Helmholtz equation. This method, akin to a reduced order model or optimal extrapolation, can reduce the norm of the residual by three to six orders of magnitude as shown in [15]. NEKBOX includes a minor modification to continuously update the projection space, which reduces the memory consumption by about 4×.

The Helmholtz problems have a moderate condition number, given their Δt^{-1} contribution to the mass matrix, so Jacobi-preconditioned conjugate gradients (CG) are sufficient. In NEKBOX' realization of the spectral element method, operators are decomposed into local operators that act on elements coupled by continuity at element boundaries, similarly to the nearest-neighbor ghost cell exchange with depth 1, independent of the spectral order, p. This facilitates very high order calculations without added communication.

Elements of different aspect ratios are built into an orthogonal mesh. A higher gridpoint density is needed to resolve the boundary layer in the near-wall region. The non-uniform span-wise mesh is computed by an offset sine profile, $x \rightarrow \sin(\alpha(1-x))/\sin(\alpha)$. The mesh is chosen such that there are: 4 points within the first wall unit, 10 points within the first 6 wall units, a maximum span-wise spacing of 5, and a maximum stream-wise spacing of 10 [18], and are given in Table I. Due to the non-uniformity of the gridpoint distribution in the span-wise direction, the time-step $\Delta t \sim \min(\Delta \mathbf{x}/|\mathbf{u}|)$ satisfying the CFL condition can be constrained, either by the spacing in the wall-normal direction, Δx_{min}, or in the stream-wise direction, Δz_{min}, depending on the flow.

N_x	N_z	p	α	Δx_{min}	Δz_{min}
128	1024	15	0.22	0.15	1.426
64	512	31	0.36	0.111	0.693

Table I
MESH PARAMETERS: N_x AND N_z ARE THE NUMBER OF ELEMENTS IN THE SPAN-WISE AND STREAM-WISE DIRECTIONS, RESPECTIVELY. p IS THE SPECTRAL ORDER AND α IS THE SINE PROFILE OFFSET. Δx_{min} AND Δz_{min} ARE THE MINIMUM GRID SPACING IN THE SPAN-WISE AND STREAM-WISE DIRECTION, RESPECTIVELY, IN WALL UNITS.

Within each element, fields are represented as tensor products of Gauss-Lobatto-Legendre (GLL) quadrature points:

$$\phi(x,y,z) = \sum_{i=0}^{p}\sum_{j=0}^{p}\sum_{k=0}^{p} \tilde{\phi}_{i,j,k,e} l_i(x) l_j(y) l_k(z), \tag{5}$$

where p is the spectral order of the method, $e(x,y,z)$ is the index of the element in the mesh, $l_i(x)$ is the ith Lagrange polynomial through the GLL points of element e, and $\tilde{\phi}_{i,j,k,e}$ is the value of the solution at the corresponding GLL point. The GLL points are chosen to avoid Runge's phenomenon while maintaining a diagonal mass matrix. The diagonal mass matrix makes CG iterations consist mostly of vector-vector operations and inner products that are aggressively coalesced, and one operator evaluation. The tensor product allows the local operator to be further decomposed into 1D operators with an arithmetic intensity that is linear in the spectral order:

$$H_e = (H_x \otimes I_y \otimes I_z) + (I_x \otimes H_y \otimes I_z) + (I_x \otimes I_y \otimes H_z), \tag{6}$$

where H_x, H_y, H_z are 1D representations of the local operator H_e and I is the identity matrix.

C. Auto-generated small matrix multiplies

The 1D matrix multiplications with an inner dimension equal to one greater than the spectral order, $p+1$, are essential to NEKBOX's performance. Although the efficient implementation of fast matrix multiplication is extensively studied, it is very challenging to speed-up the small variants. Previously, NEKBOX relied on an internal FORTRAN-based matrix-matrix implementation, called here `mxm_std`. This library explicitly defines multiple interfaces corresponding to values of the inner dimension k, and provides unrolled FORTRAN primitives to the compiler. However, `mxm_std` does not deliver reasonable performance on modern Intel CPU architectures [16], particularly at higher orders, and thus high arithmetic intensities. Intel's own Math Kernel Library (MKL) is also unable to achieve a reasonable performance for the small general matrix multiplications (GEMMs) found in NEKBOX [16].

To improve the performance, we have integrated LIBXSMM into NEKBOX. This library creates a specific kernel implementation for each small matrix multiplication size and optimizes that kernel specifically for available vector extensions. LIBXSMM does not need to be configured for a specific target architecture. Instead, it leverages the just-in-time (JIT) compilation technique to generate requested kernels automatically at runtime. Kernels are cached to amortize generation costs over the application run-time, and NEKBOX stores pointers to common kernel sizes to bypass look-up overhead. It also offers special copy routines that leverage non-temporal stores. The Intel architecture system generates read-for-ownership (RFO) instructions on writes, due to cache coherency policies. Non-temporal stores bypass the cache and therefore reduce the superfluous RFO traffic when the results would not have been read before falling out of cache.

D. Global coarse problem

Unlike the Helmholtz problems, the Poisson problem is ill-conditioned and will not rapidly converge with Jacobi preconditioned CG [14]. Instead, the Poisson problem is solved with the more robust generalized minimum residual method (GMRES) aggressively preconditioned with a hybrid Schwarz multigrid method (HSMG), c.f. [14]. HSMG performs two multigrid steps within each element, reducing the problem size by an aggregate factor of p^3, where p is the spectral order. At each of those steps, an additive overlapping Schwarz preconditioner exactly solves the restricted Poisson problem. Then, only one degree of freedom per element remains, defining a global coarse problem.

The global coarse problem can be solved exactly via Cholesky factorization with nested dissection (XXT) as described by [19], or, in the case of uniform meshes, Fourier methods. In this study, we use an algebraic multigrid (AMG), c.f. [20], coarse solve designed for Nek5000. Rather than focus on improving the performance of the AMG, which is already established, we aim to make the global problem as small as possible. Because the size of the coarse problem is independent of the spectral order, by using very high order elements we reduce the size of the coarse problem for fixed degrees of freedom (DOFs). For example, at $p = 31$ a 69 billion grid point problem has a coarse problem with only 2.1 million DOFs. The dramatic reduction in problem size improves the scalability and time to solution despite very high order problems being more arithmetically intensive and more accurate.

E. Lossless I/O compression

I/O performance is a key ingredient to do gradient based optimization through adjoint models. Computed in reverse order of the primal computation, the solution at every time-step must be recovered for each adjoint time-step, requiring a careful combination of memory check-pointing, disk check-pointing, and recomputation. Nek5000 has recently incorporated a novel adjoint check-pointing scheme that uses all system resources, memory, disks, and archive, proving that adjoint computations of nonlinear problems and are feasible at large scale [21].

I/O serves many purposes, such as visualization, resilience and adjoint check-pointing. Sophisticated techniques, *e.g.* example burst buffers, have been adapted to the I/O usage patterns of large scale simulation codes. A supplement to improving the hardware is to directly reduce the transfer size through data compression, which both increases effective data throughput and reduces the final storage requirement on the disk. The I/O implementation of NEKBOX is laid out in Figure 2. It distinguishes *I/O ranks* as a subset of *compute ranks*. For brevity reasons, we only focus on the write operation, which is fundamentally similar to reads and more frequent. After a write operation is initiated, every compute rank sends its data to the I/O ranks, which then dispatch the data to the system I/O API, *i*.e. `fwrite`.

We have chosen the lossless lz4 compression algorithm, an extremely fast compression algorithm used for online RAM compression as analyzed by [22]. By compressing the data before sending them to I/O ranks, not only is the cost of compression parallelized over all the compute ranks but also the network load is also reduced. The compute rank compression structure is very generic and may be applied to other large scale simulation codes. Currently, we are working on the implementation of a lossy compression tailored to the solutions of PDEs.

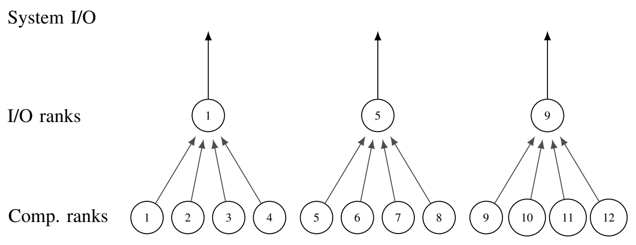

Figure 2. I/O write operation using 12 processes with 3 I/O ranks.

III. Performance Experiments

The optimizations performed here are applicable to traditional homogeneous architectures with cache hierarchies. In particular, LIBXSMM already supports not only the AVX2 instructions set on Haswell, but also the AVX-512 instructions set available on the Intel Xeon Phi processor (KNL). The executable used here is in fact binary compatible with the Intel processors and we will demonstrate that this and other on-node optimizations on Haswell and KNL Cray XC40 systems.

A. Performance analysis

Our goal is to establish a holistic, absolute metric for application performance. It is the authors' view that the ideal metric would be the share of instantaneous rate limiting resource, integrated over the entire application. The problem with this metric is that the rate limiting resource varies over the execution of the application and is difficult to track. Fortunately, we have found NEKBOX to be predominantly memory-bandwidth bound, so we treat memory-bandwidth as rate-limiting through the application. Using an on-node metric has the added advantage of capturing communication overhead and parallel efficiency.

Floating point and memory operations are counted explicitly within the code. The counters are intended to represent the workload of the algorithm, not necessarily the actual operations performed by the hardware. For example, read-for-ownership is not counted as a memory operation, as it is a byproduct of the cache policy. Cache is assumed to be larger than an element and smaller than a field, and only memory operations that could not fit into cache are counted. For example, successive matrix-multiplies that use the same transformation matrix (i.e., the basis transformation used in the iterative solvers) only count the load of such a matrix once. These choices make the reported operation counts lower bounds of number of operations evaluated by the hardware. The advantage is that these bounds do not reward inefficient use of instructions. In addition, we collect solution time and power consumption measurements.

Our primary performance metric is overall efficiency, in this case measured by the share of maximum memory bandwidth, measured by the STREAM benchmark [23]. The overall efficiency metric includes a number of independent effects, *e.g.* parallel efficiency, and is a lower bound for each of them. The overall efficiency is also absolute, in that it defines an upper bound for the available remaining optimization. An application with a parallel efficiency e can not be optimized by more than a factor of $1/e$ without changing the workload.

This is the primary motivation for the overall efficiency metric: when performance tuning, it is easy to tell how much improvement has been made, but hard to tell how much is left. By establishing a bound on what is left, our efforts can be directed towards areas of greater potential improvement. Indeed, it is this process of optimizing low efficiency implementations, rather than simply hot-spots, that led to most of the performance improvements described here.

B. Platforms

Our goal is to demonstrate performance reproducibility or sensitivity on three Cray XC40 machines, two with slightly different sizes and network configurations, and one with a new architecture. The results shared in the paper are considering the average value of four similar runs (with 512 time steps) not in a dedicated mode, except for the largest runs on ShaheenII XC40 using 6144 nodes. Outliners runs have been removed when intensive I/O on the parallel file system or network throttling have been observed in the systems. On all systems we used Intel's compilers and Cray's MPI implementation. Details on the installation and reproducibility are available in the Appendix.

1) ShaheenII XC40: The ShaheenII XC40 - supercomputer [24] installed at King Abdullah University of Science and Technology (KAUST) and managed by the KAUST Supercomputing Lab is a 36-cabinet Cray composed of 6,174 dual socket compute nodes based on 16 core Intel® Xeon® E5-2698v3 (code-named Haswell) processors running at 2.3GHz.?de Each node is equipped with 128GB of DDR4 memory running at 2,300MHz. The 36 cabinets are connected via the Aries high speed network on a dragonfly topology capable of achieving 57% of the maximum global bandwidth between the 18 groups of two cabinets. Overall the system has a total of 197,568 processor cores and 790TB of aggregate memory, and it is capable of delivering a theoretical peak of 7.2 PFLOP/s.

2) Trinity XC40 : The runs on Trinity were executed only on Trinity Phase-1, a Cray XC40 production system for the National Nuclear Security Administration (NNSA), is managed in a joint effort of the New Mexico Alliance for Computing at Extreme Scale (ACES) between Los Alamos and Sandia national laboratories. It features the same nodes as ShaheenII, but 9,436 of them resulting into a total of 301,056 cores and over 1.2 PB of aggregate memory with a Phase - 1 achieved a performance of 8.1 PFLOP/s with HPL, as reported in the TOP500 list in November 2015, while delivering a theoretical peak of around 11 PFLOP/s.

Data per core	Unit	Min	Max	Average
HPL	[GFLOP/s]	29.2	31.9	30.6
STREAM TRIAD	[GB/s]	3.3	3.8	3.7
nektester, $s = 1$	[GFLOP/s]	31.7	36.2	34.9
nektester, $s = 768$	[GB/s]	2.6	3	2.8
MPI allreduce (1 double)	[μs]	130.6	130.7	130.6
Gather Scatter	[μs]	571.4	1686.5	1431.5

Table II
HEALTH CHECK OF SHAHEENII XC40.

Data per core	Unit	Min	Max	Average
nektester, $s = 1$	[GFLOP/s]	31.8	35.5	33.7
nektester, $s = 768$	[GB/s]	2.6	2.8	2.7
MPI allreduce (1 double)	[μs]	83.5	83.5	83.5
Gather/Scatter	[μs]	354.7	914.9	651.1

Table III
HEALTH CHECK OF TRINITY XC40.

In contrast to ShaheenII XC40, Trinity XC40's network achieves 22% of Aries theoretical bandwidth. The default programming environment on Trinity XC40 is Intel, using Intel 16 with MPICH 7.3.1 under CLE6 UP01 OS.

Cori XC40 is the US National Energy Research Scientific Computing Center's newest supercomputer. It features two different kinds of nodes: 2004 Intel® Xeon® "Haswell" processor nodes and 9304 Intel® Xeon Phi™ (code-named Knights Landing) nodes. On Cori, each Xeon Phi node has 68 cores with support for 4 hardware threads each, 96 GB DDR4 2400 MHz memory and six 16GB DIMMs (115.2 GB/s peak bandwidth). The total aggregate memory (combined with MCDRAM) is 1 PB. All runs on Cori are executed on the KNL partition and are done under CDT16.10, using Intel 17 and MPICH 7.4.4.

C. Acceptance benchmarks, regression testing

To achieve outstanding performance on highly coupled HPC applications, it is necessary to detect faulty components that lead to even minor performance regressions. This need has grown with the complexity of today's HPC systems. In

Data per core	Unit	Min	Max
nektester, $s = 1$	[GFLOP/s]	29.5	32.8
nektester, $s = 768$	[GB/s]	1.6	2
MPI allreduce (1 double)	[μs]	10.5	10.5

Data per core	Unit	Min	Max
nektester, $s = 1$	[GFLOP/s]	29.7	34.1
nektester, $s = 768$	[GB/s]	1.7	2
MPI allreduce (1 double)	[μs]	10.5	10.5

Table IV
HEALTH CHECK OF CORI XC40 (MCDRAMCACHE MODE, TOP) AND CORI XC40 (MCDRAM FLAT MODE, DOWN)

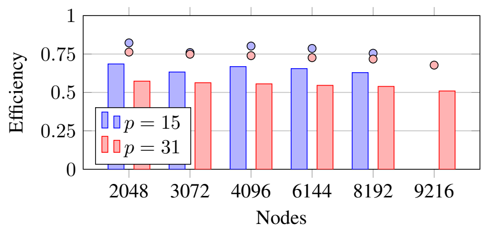

Figure 3. Overall efficiency of memory bandwidth for weak scaling on Trinity XC40: 262,144 grid points per core. Points show parallel efficiency based on 1 node run at $Re_\tau = 2000$.

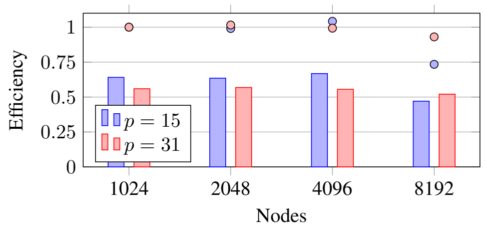

Figure 4. Overall efficiency of peak memory bandwidth for strong scaling on Trinity XC40: 34 billion grid points total. Points show parallel efficiency based on 1024 node run, which is the smallest for which the problem fits in memory.

this study, before conducting the large scale runs, we performed an extensive regression testing [25] using synthetic benchmarks designed to emulate the computational demands of NEKBOX.

Therefore, before starting the full scale campaign and extracting the performance numbers, regression testing using synthetic benchmarks, like single node HPL, stream and Nektester have been executed on all available nodes of Cray XC40 systems. Nektester is a benchmark developed to extract the performance of LIBXSMM library and the MPI allreduce. All synthetic benchmarks reported the best set of nodes along with the weak ones that have been excluded for the performance application executions. The performance per core of the vetted systems are shown in Table II,Table III and Table IV, for ShaheenII, Trinity and Cori respectively.

The parameter s represents the size of the data ($s = 768$ is bigger than the cache). The regression testing shows the variation across the whole system on the peak compute and peak memory bandwidth. The relative percentage difference between the best and weak nodes can reaches up to 15% on ShaheenII and around 11% on Trinity . The results of LIBXSMM with nektester exhibits higher performance in FLOP than HPL since it runs small matrix multiplications out of L1 cache. Indeed, using 6144 nodes, 6.9 PFLOP/s has been achieved with nektester, whereas we reached 5.53 PFLOP/s with HPL.

Although the compute intensive and memory bandwidth results on ShaheenII and Trinity are quasi-similar, we observed that for the MPI allreduce ShaheenII is around 60% slower than Trinity. These results were achieved on ShaheenII using CLE 5.2 Up 04 OS, and all the performance runs on Trinity were achieved with an under-populated network and using the CLE 6.0 Up 01 withCray MPICH 7.3.1. Using the same MPICH version on Shaheen did not improve the performance of collective operations. The lower performance of ShaheenII is counter to our expectation, since 22% of Aries theoretical bandwidth can be achieved on Trinity and it's 57% on ShaheenII.

IV. Performance Results

A. Scaling

We can better bound the parallel efficiency by varying the scale while keeping the other properties of the problem fixed. This is much like the standard weak and strong scaling techniques, but with parallel efficiency based on changes in the overall efficiency rather than run time. For weak scaling, this helps correct for changes in the profile induced by the different size problems. In particular, larger problems may take more iterations to converge, increasing their overall cost. In NEKBOX, these effects are independent of the number of ranks. Increasing the number of ranks for a fixed problem size, i.e. strong scaling, does not effect the counted workload, so measuring the efficiency corrects for changes in the workload due to physical effects.

The results of weak and strong scaling studies on Trinity XC40 - Phase-1 are presented in Figure 3 and Figure 4, respectively. Weak scaling at 262,144 grid points per rank is steady from 2048 to 8192 nodes at 92% and 94% for $p = 15$ and $p = 31$, respectively. Strong scaling is also steady from 1024 to 4096 ranks, at 104% and 99%, but drops at 8192 ranks with 131,072 grid points per rank to 73% and 93%, for $p = 15$ and $p = 31$, respectively. The super-scaling for $p = 15$ is likely due to contention between multiple running smaller jobs on a shared resource, whereas larger jobs are more likely to be scheduled exclusively. The $p = 15$ case is much more sensitive to strong scaling than the $p = 31$ case, due to the larger global coarse problem size. At 132,072 grid points per rank, the higher order $p = 31$ case is more efficient than the $p = 15$ case, despite having a significantly higher arithmetic intensity.

On 294,912 cores of Trinity with $p = 31$, we achieve an overall memory bandwidth of 516.5 TB/s. Although not rate limiting, the floating point performance of that run was 1.05 PFLOP/s.

B. Portability

It is our intention to demonstrate portability with a comparison of ShaheenII XC40 and Trinity XC40. ShaheenII XC40 has fewer nodes, but more network links per node.

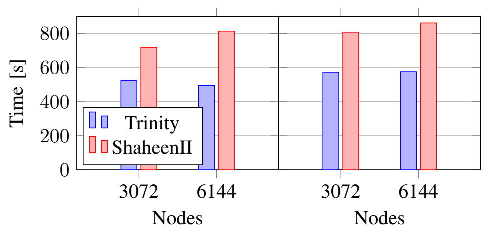

Figure 5. Comparison of Trinity XC40 to ShaheenII XC40 for weak scaling with 262,144 grid points per core. Left part order $p = 15$; right part order $p = 31$.

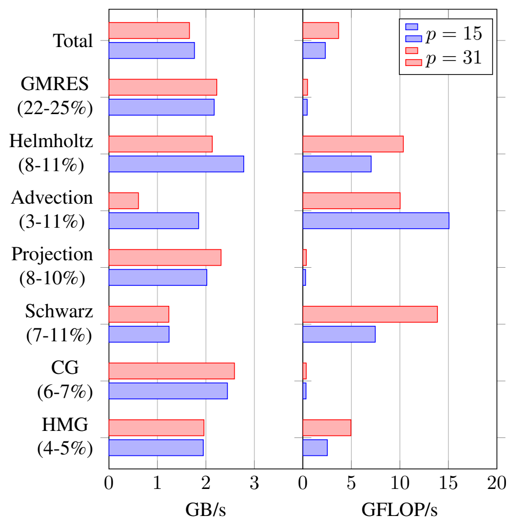

Figure 6. Performance per core of key code sections for fully developed turbulent flow at $p = 15$ and $p = 31$ on 8192 nodes of Trinity XC40.

Thus, we expect ShaheenII to outperform Trinity for runs of the same scale.

We duplicated four runs on both machines: weak scaling 3072 and 6144 nodes cases at both $p = 15$ and $p = 31$. The overall efficiency for the four cases on Trinity and ShaheenII are shown in Figure 5.

Contrary to our expectations, in these cases, ShaheenII is 40% to 65% slower than Trinity, with the largest gap for 6144 nodes and $p = 15$.

The degraded performance on ShaheenII is similar to the synthetic results presented in the previous section. The improvement analysis and performance improvement are discussed at the end of this section.

It is worth highlighting the fact that the scaling on Trinity, shown in Figure 3 and Figure 4 is achieved with an under-populated network.

C. On-Node Performance

A breakdown of the performance of key code regions, defined exclusively, is given in Figure 6. GMRES is the Poisson solver, CG is the Helmholtz solver, Helmholtz is the local operator evaluation in both cases, advection is the de-aliasing of the advection operator on the RHS, Schwarz is the local part of the Poisson preconditioner, HMG represents the local restriction and interpolation operators in the element-wise multigrid, and Projection corresponds to the pre- and post-solver historical projection steps. All sections are exclusive, *e.g.* GMRES excludes time from the operator and preconditioner calls that it makes.

NEKBOX spends the majority of its time in local, memory bandwidth intensive workloads. However, there are three floating-point intensive regions of the code that suffer when floating point efficiency is low: the Helmholtz operator evaluation, the local Schwarz solves in the preconditioner, and the evaluation of the nonlinear advection operator.

The sensitivity of these regions to floating point efficiency is easily demonstrated by turning off the LIBXSMM library. Figure 7 shows the FLOP rates for the three most compute-intensive regions, with and without enabling the library. In each case, it provides around 2× improved performance, compared to compiled native FORTRAN implementations. The effect on the total application performance is damped by communication and vector-vector workloads that do not use LIBXSMM.

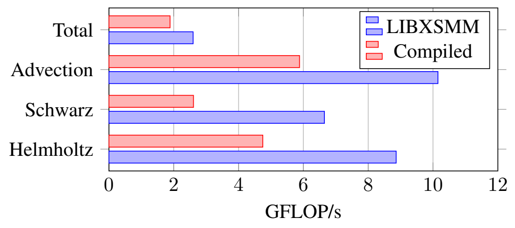

Figure 7. FLOP rates of compute-intensive code sections for $p = 31$ comparing LIBXSMM and compiled native FORTRAN on 6144 nodes ShaheenII XC40 for fully developed turbulent flow.

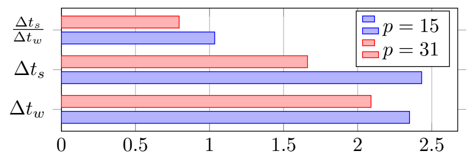

Figure 8. Time to solution metrics comparing $p = 15$ to $p = 31$ for a developed flow on 8192 nodes of Trinity XC40. Δt_w is the wall-clock time per time-step in seconds, and Δt_s is the simulation time per time-step in μ-seconds.

D. Time to Solution

To compare the performance of the two orders, $p = 15$ and $p = 31$, we first evolve the 8192 nodes cases to $\mathrm{Re}_\tau \approx 2000$. This ensures that the flow fields have the same structure as the fully developed flows from which detailed statistics are gathered, and that the majority of compute time is spent. There are two factors that affect the rate at which turbulent statistics are gathered: the size of the time-step and the computational cost of each of those time-steps. Together, they can be composed into a simulated time per wall-clock

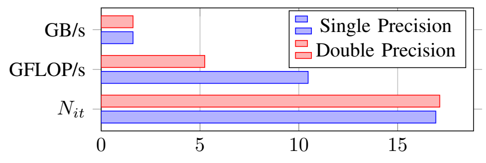

Figure 9. Memory, floating point, and average number of iterations, N_{it}, in single and double precision, performed on 8192 nodes of Trinity XC40.

time metric, for which larger is better. These three metrics are presented in Figure 8.

The smallest inter-point spacing in GLL elements goes with the square of the spectral order p. Therefore, when we roughly double the spectral order, while keeping the total number of grid-points fixed, we expect the smallest inter-point spacing to decrease by a factor of 2. However, in the span-wise direction the elements are packed near the boundaries to resolve the boundary layer. The resolution rules for doing so are more easily satisfied by higher order elements, resulting in element aspect ratios closer to 1. In this case, the smallest inter-point spacing decreases by a factor of about 1.35. Indeed, the ratio of the time-steps is 1.46, as opposed to 2 for a uniform mesh.

The $p = 31$ case has a higher local cost due to an increasing arithmetic intensity in the operator's evaluation, but it also has an 8x smaller global coarse problem. The balance of these two factors depends on the balance between the difficulty of the Poisson and Helmholtz problems. For flow at $\mathrm{Re}_\tau \approx 2000$, this balance is in favor of $p = 31$. Despite being higher order, $p = 31$ is actually faster on a per time-step basis. This speedup is problem and scale dependent, but counter-intuitive in any case.

These two effects are composed to get the total relative time-to-solution. The time-step stability effect is stronger, so $p = 15$ is able to generate statistics 30% more quickly. However, this analysis does not include accuracy and the $p = 31$ case is more accurate. In principle, the improved accuracy could be traded for fewer degrees of freedom, reducing the time to solution of the $p = 31$ case. The degree to which this trade-off can be realized, particularly for internal flows, requires further study.

E. Mixed Precision

The preconditioner is approximate by design and therefore a strong candidate for mixed precision. In the case of preconditioning, the cost of reduced precision would be a change in the number of iterations necessary for convergence, because the convergence test itself is performed in double precision. Another concern is that the reduced load will decrease efficiency, in this case via a drop in memory bandwidth. Finally, the gain in performance can be seen directly by an increase in the FLOP rate.

These metrics are presented in Figure 9. We find that single precision preconditioning has an insignificant effect

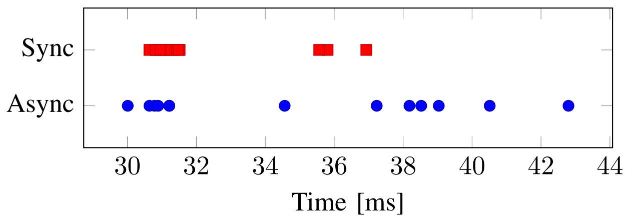

Figure 10. Distribution of runtime per CG iteration on 1024 nodes of Trinity XC40. Runs were performed under a variety of third party machine loads.

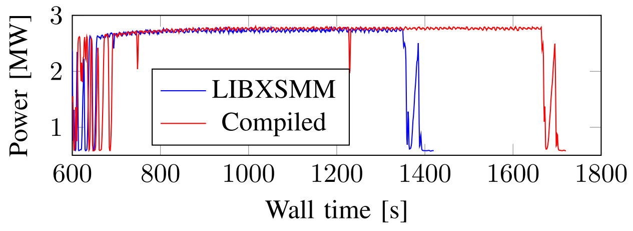

Figure 11. Power consumption on ShaheenII for $p = 31$ comparing LIBXSMM and compiled native FORTRAN on 6144 nodes ShaheenII XC40.

on the number of iterations and the bandwidth. Therefore, our single precision preconditioner achieves a $2\times$ improvement in FLOP rate and total run-time. All other performance measurements are taken with single-precision preconditioning. The preconditioner accounts for roughly 15% of the total runtime. Therefore, had we performed it in full double precision, the runtime would haved increase by roughly 15% and scalability would have improved.

F. Asynchronous Communication

The effect of the asynchronous nearest-neighbor communication is negligible compared to the whole performance of the application, so we measure the time per CG iteration, which has the highest density of such calls. Distributions of timings taken from 1024 nodes runs on Trinity are shown in Figure 10. The runs are identical and the spread in time can be attributed to the contention on the network due to third-party jobs on other nodes in the machine. We claim the performance signal is best seen in the floor of these timings, which is 30.01 ms with asynchronous and 30.64 ms without, for a 2% speed-up. The improvement at this scale on Trinity is negligible, but this is somewhat expected given that only a streaming overlap of communication and computation are possible, due to the dependency structure of the solver. To achieve a more substantial improvement, the methods would need to be changed to expose task parallelism at coarser granularity, which may not be a priority given NEKBOX's existing scalability.

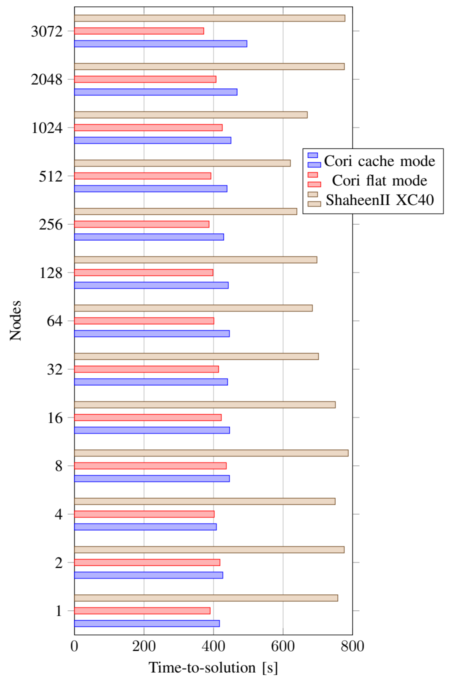

Figure 12. Comparison of Cori XC40 with ShaheenII XC40 for time-to-solution; 131,072 grid points per core and $p = 15$; 512 time steps for fully developed turbulent flow; I/O with 27 statistics is included.

G. Energy Consumption

Figure 11 plots the power consumption of the entire ShaheenII XC40 system for a 6144 nodes simulation, including cabinets, blowers, etc., extracted from the Cray Power Measurement Data Base for one run with LIBXSMM enabled and one run without. LIBXSMM shortens the execution time of the simulation without increasing the power consumption of the supercomputer, which is consistent with the consumption of LINPACK. Therefore, enabling it provides an energy saving factor that corresponds to the reduction in wall-clock time. In this particular instance, it reduces total energy consumption of the solver, as measured over the central plateau, from 761.9 kWh to 532.59 kWh, for 229.3 kWh difference. This amounts to energy savings of approximately 30%, using around 2 to 3% less peak power(the maximum achievable power consumption during the run).

H. Performance on KNL

On Cori XC40, we used NEKBOX to simulate a slightly scaled down version of the square duct problem with 131,072 grid points per rank and 64 ranks per socket, using order 15. The code was not modified in any way. Only the JIT compiler in LIBXSMM transparently enabled the AVX-512 instruction set.

Fig. 12 plots the time-to-solution for 512 time steps on ShaheenII XC40 and Cori XC40, in both flat and cache mode. All simulations were performed using $p = 15$ and a fully developed turbulent flow at $Re_\tau = 2000$ with 27

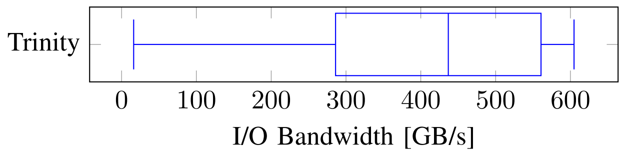

Figure 13. Distribution of I/O bandwidths on 8192 nodes of Trinity XC40 during heavy third party disk usage.

production run statistics written to file every 500 time steps. Fig. 12 shows that NEKBOX scales up to 3072 Xeon Phi nodes with an excellent parallel efficiency.

Both flat and cache modes on Cori significantly outperform ShaheenII on a node-by-node basis. Flat mode outperforms cache mode in all cases, because it is able to guarantee the use of MCDRAM. The performance gap between cache and flat mode grows with the scale of the run. This can be attributed to the design of the cache: the direct mapped model performs poorly when physical pages are fragmented, due to previous jobs. The likelihood of a fragmented node increases with the total number of nodes.

For the largest run on 3072 nodes of Cori in flat mode, we achieved 378 TFLOP/ with an aggregated bandwidth of 310 TB/s which is corresponds to a 2.11× faster time-to-solution in comparison with the same number of nodes on ShaheenII. On a task-by-task basis, Cori outperforms ShaheenII by 3.4× in the Helmholtz solver, 1.8× in the overlapping Schwarz solver, and 3.7× in the advection operator. The relatively poor performance of the Schwarz solver is attributed to the memory misalignment forced by the single ghost region in the solver, compared to easily aligned accesses in the Helmholtz and advection portions of the program.

I. I/O and Lossless Compression on Lustre

Our runs on Trinity were conducted just before the machine was taken offline for classified work. Therefore, there was an uncharacteristically large load of third-party I/O on the filesystem that added substantial noise to I/O readings. Therefore, I/O has not been included in the total performance timings, and have presented I/O results from Trinity separately.

For turbulent calculations such as this one, statistics are accumulated in-situ and disk I/O disk is used primarily to checkpoint. The checkpoint frequency is generally based on machine stability, and can reasonably be set to hourly on ShaheenII or Trinity XC40. The size of the checkpoint is 5 fields, or 40 bytes per grid point. For the 8192 node case, this corresponds to 2.75 TB per checkpoint. Figure 13 shows the distribution of I/O timings on the noisy Trinity system. On a dedicated system with I/O isolation, we would expect near the peak of these rates which has been achieved on ShaheenII during dedicated runs on Lustre setting to the maximum stripe count of 144 to achieve around 460 GB/s

90% of the IOR write performance.

To further improve the performance, we demonstrate the potential advantage of lossless compressed I/O. Measurements performed on coarsened data at $p = 7$ achieved a compression ratio of 1.09 evenly spread across the duct, indicating that the flow is evenly turbulent. At higher order and in less turbulent flows, the compression ratio is expected to increase, making 1.09 a lower bound. System I/O noise had a variation well beyond 9% such that we cannot discern any performance effects. Nonetheless, a 9% reduction in storage requirements is significant at scale.

J. Burst Buffer performance

ShaheenII is also composed of 268 Cray DataWarp (DW) accelerator nodes hosting a total of 536 Intel P3608 SSD cards, providing an aggregate capacity of 1.56 PB. Each Burst Buffer node provides aggregated peak write/read bandwidth 6 GB/s and 10 GB/s, respectively. Using the IOR benchmark on 268 DataWarp nodes and 5628 compute nodes we achieved 2 TB/s and 1.6 TB/s in the IOR read and write, respectively.

Burst Buffer resources are managed on ShaheenII with SLURM and can be requested within the SLURM script using #DW directive. These directive are used to stage asynchronously files from Lustre to the burst buffer or stage files out from burst buffer to Lustre.

Figure 14 shows the performance betweeen ShaheenII obtained either with a Lustre or Burst Buffer for a time-to-solution test case using 131,072 grid points per core and $p = 15$; 512 time steps for fully developed turbulent flow and storing statistics per grid point.

Using 3072 compute nodes (32 MPI tasks per node), we have in total 12.8 Billions grid points generating large number of files for a volume of 4.5 TB. After identigying the best balance between the number of compute and DataWarp nodes, we were able to write on ShaheenII Burst Buffer 3.11× faster than on the Lustre one. Indeed, using 128 DataWarp nodes, the peak write performance that we achieved was 560 GB/s, wheras a maximum of 180 GB/s in writing on ShaheenII parallel file system was obtained. Consequently, the time to solution required for the application to complete was reduced, while also increasing the overlap of computation with I/O, typically reducing the application elapsed time by up to 23% with the 12.8 Billions grid point test case using 3072 compute nodes. For the test case with 256 nodes and a total of 375 GB of data, and by using 2 DataWarp nodes, the Burst Buffer was only 3% faster than Lustre.

K. Effect on the programming environment and regression testing

The low performance of ShaheenII compared to Trinity lead us to investigate and determine root causes. Even though these 2 systems have different operating systems,

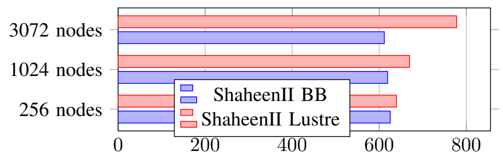

Figure 14. Comparison of ShaheenII XC40 on Lustre and Burst Buffer for time-to-solution including I/O.

the main differences, as stated in Section 3.C and Section 4.B, are on the MPI collective operations. Trinity runs were linked with the default Programming Environment 16.01, using cray-mpich/7.3.1 on the Operating System CLE 6.0 UP01 wheras ShaheenII default was cray-mpich/7.2.4 on CLE5.2 UP4. Loading a similar programming environment and linking with the same cray-mpich version did not improve the performance significantly. Upgrading ShaheenII to CLE 6.0 was scheduled only in early 2018. The HSN on Shaheen is configured with 8 optical network connections between every pair of cabinets, wheras on Trinity, the HSN is configured with 2 optical network connections. Therefore, we suspected there was an issue in the network. In order to isolate the link not performing as expected, we used the test_links tool developed at Cray that was specifically redesigned for Cray XC systems. This tool is now part of the regular regression testing protocol after every maintenance or unscheduled downtime [25].

Test_links evaluate the bandwidth of all interconnect links in any allocation of nodes and identify links with lower (by a specified amount) than the best or the average bandwidth for links of the same type. Bandwidths for each link are also recorded in tables for comparison between sets of results obtained at different times, so that one may determine whether and how an individual link performance has changed over time. The test_links code is a user-level topology-aware MPI program in which the physical location of the hardware of each process (MPI rank) in the allocation is obtained from Cray MPI library calls. The bandwidth of any link is determined by timing MPI calls involving messages of a specified size using an intensive communication pattern between ranks located on either end of that link.

Running the tests before and after the full scale runs revealed that a particular subset of nodes for which the PCIe connections to the Aries routers exhibited an unexpectedly high variability. Over time, this test revealed several links that needed to be replaced, or, in some cases, only a reboot of the nodes in the implicated blade was necessary to resolve the issue.

As detailed in [26] with the new version Cray MPI implementation available in cray-mpich/7.4.0 (it provides a significant improvement of the collective operations), the MPI_allreduce test is much faster, down to 86.9μs (very close to Trinity values); it used to be 130μs, as reported in TABLE II.

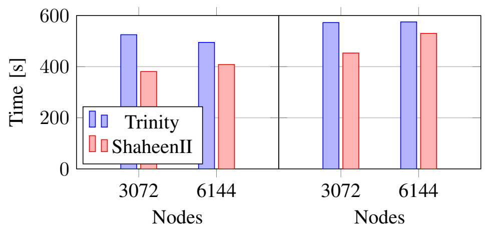

Figure 15. Updated comparison of Trinity XC40 with ShaheenII XC40 for weak scaling with 262,144 grid points per core. Left part order $p = 15$; right part order $p = 31$.

We reproduced the four runs presented in 4.A on both machines and weak scaling 3072 and 6144 node cases, at both $p = 15$ and $p = 31$, using CDT16.10. The overall efficiency for the four cases on Trinity and ShaheenII are shown in Figure 15. As expected, ShaheenII is 10% to 28% faster than Trinity in these cases, with the largest gap of 28% for 3072 nodes and $p = 15$.

V. Interpretation and Conclusions

We have used NEKBOX to perform extreme scale direct numerical simulations of the incompressible Navier–Stokes equations in long square ducts, at a bulk Reynolds number of 100,000 with a stream-wise extent of $48h$ for 69 billion grid points per field. The late-time flow field, Figure 1,is used as a starting point for a more thorough study of the secondary flow and mixing.

Simulations sustained over 1 PFLOP/s and 500 TB/s memory bandwidth on 9216 nodes of Trinity XC40, with an overall efficiency greater than 50%. The overall efficiency, measured here as the average memory bandwidth divided by the STREAM bandwidth, includes parallel efficiency, instruction efficiency, and bookkeeping overheads. It would therefore not be possible to improve NEKBOX's performance more than 2× without reducing the workload of the underlying methods.

Strong scaling shows 73% and 93% parallel efficiency at $p = 15$ and $p = 31$, respectively, from 1024 to 8192 nodes on Trinity XC40. Weak scaling at 262,144 grid points per core shows a steady performance at approximately 75% parallel efficiency, through 9216 nodes on Trinity XC40. The improved scalability for $p = 31$ is due to a smaller global problem size. For the developed science case, $p = 31$ was 10% less expensive on a per time-step basis. This is possible only with highly efficient small matrix multiplications, as implemented in LIBXSMM, which speeds up the compute intensive code regions by 2× for $p = 31$. Nonetheless,an improved efficiency does not lead to higher power usage. Shorter runtimes with LIBXSMM translate into a reduction in energy consumption, by up to 30%. Improving implementations to approach parity between arithmetic and memory workloads, with respect to hardware capabilities, can not only reduce time to solution but also total cost.

Single precision preconditioning has an insignificant effect on the iteration count but improves FLOP rates by 2× while maintaining memory bandwidth.

NEKBOX achieves scalability on all three XC40 systems with a pure-MPI programming model. Without special tuning, NEKBOX on Cori achieved up to a 2.11× speedup per node vs ShaheenII XC40 for $p = 15$ and 131,072 grid points per core. NEKBOX scales up to 3072 Xeon Phi nodes with perfect parallel efficiency.

By using Burst Buffer on DataWarp (thanks to the #DW directive available with SLURM), the writing rate was increased by over 3× and reduced by 23% the elapsed time to solution compared to data stored in the parallel Lustre file system.

In addition, we detailed the effect of programming environment versions and the importance of the regression testing, which can be critical in terms of applications' performance. Regular comparison on standard benchmark shall be shared between HPC centers managing similar architecture systems in order to detect issues in the hardware or software environment, and to allow the release of a more reliable and performing HPC programming environment on the Cray XC40 and beyond systems to the users.

All of the contributions presented in this work have the potential of benefiting the wide community of scientists and researchers in computational fluid dynamics and other PDE-based numerical fields.

VI. Acknowledgments

For computer time, this research used the resources of the Supercomputing Laboratory at King Abdullah University of Science and Technology (KAUST) in Thuwal, Saudi Arabia; 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; and the Trinity project managed and operated by Los Alamos National Laboratory and Sandia National Laboratories.

References

[1] R. Vinuesa *et al.*, "Aspect ratio effects in turbulent duct flows studied through direct numerical simulation," *J. Turbul.*, vol. 15, no. 10, pp. 677–706, 2014.

[2] A. Noorani *et al.*, "Particle transport in turbulent curved pipe flow," *J. Fluid Mech.*, vol. 793, pp. 248–279, 2016.

[3] A. Lozano-Durán and J. Jiménez, "Effect of the computational domain on direct simulations of turbulent channels up to Re_τ = 4,200," *Phys. Fluids*, vol. 26, no. 1, 2014.

[4] J. Slotnick *et al.*, "CFD vision 2030 study: a path to revolutionary computational aerosciences," Tech. Rep. NASA/CR-2014-218178, 2014.

[5] Hutchinson, M. *et al.*, "Nekbox," 2016. [Online]. Available: https://github.com/maxhutch/NekBox

[6] P. Fischer *et al.*, *Nek5000 User Documentation*. Argonne National Laboratory, 2016. [Online]. Available: https://nek5000.mcs.anl.gov/documentation/

[7] A. Pinelli *et al.*, "Reynolds number dependence of mean flow structure in square duct turbulence," *J. Fluid Mech.*, vol. 644, pp. 107–122, 2010.

[8] P. R. Spalart, "Strategies for turbulence modelling and simulations," *Int. J. Heat Fluid Fl.*, vol. 21, no. 3, pp. 252–263, 2000.

[9] D. Rossinelli *et al.*, "11 PFLOP/s simulations of cloud cavitation collapse," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–13.

[10] A. Heinecke *et al.*, "Petascale high order dynamic rupture earthquake simulations on heterogeneous supercomputers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 3–14.

[11] H. M. Tufo and P. F. Fischer, "Terascale spectral element algorithms and implementations," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ser. SC '99. New York, NY, USA: ACM, 1999.

[12] C. Canuto *et al.*, *Spectral Methods in Fluid Dynamics*, ser. Springer Series in Computational Physics. Springer Berlin Heidelberg, 1988.

[13] M. O. Deville *et al.*, *High-Order Methods for Incompressible Fluid Flow*. Cambridge University Press, 2002, cambridge Books Online.

[14] P. F. Fischer and J. W. Lottes, "Hybrid Schwarz-multigrid methods for the spectral element method: Extensions to Navier–Stokes," in *Domain Decomposition Methods in Science and Engineering*, T. J. Barth, Ed. Springer, 2005, pp. 35–49.

[15] P. F. Fischer, "Projection techniques for iterative solution of Ax = b with successive right-hand sides," *Comput. Method. Appl. M.*, vol. 163, no. 1-4, pp. 193–204, 1998.

[16] M. Hutchinson *et al.*, "Efficiency of high order spectral element methods on petascale architectures," *Proceedings of the International Supercomputing Conference, (ISC'16)*, 2016.

[17] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "Libxsmm: Accelerating small matrix multiplications by runtime code generation," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 84:1–84:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3015017

[18] D. Krasnov *et al.*, "Numerical study of magnetohydrodynamic duct flow at high Reynolds and Hartmann numbers," *J. Fluid Mech.*, vol. 704, pp. 421–446, 2012.

[19] H. M. Tufo and P. F. Fischer, "Fast parallel direct solvers for coarse grid problems," *J. Parallel Distr. Com.*, vol. 61, no. 2, pp. 151 – 177, 2001.

[20] J. Lottes, "Independent quality measures for symmetric algebraic multigrid components," *Argonne National Laboratory, Mathematics & Computer Science Division*, 2005. [Online]. Available: http://www.mcs.anl.gov/papers/P1820-0111.pdf

[21] M. Schanen, O. Marin, H. Zhang, and M. Anitescu, "Asynchronous two-level checkpointing scheme for large-scale adjoints in the spectral-element solver Nek5000," *Procedia Computer Science*, vol. 80, pp. 1147–1158, 2016.

[22] Y. Collet, "Lz4: Extremely fast compression algorithm," *code. google. com*, 2013. [Online]. Available: http://cyan4973.github.io/lz4/

[23] J. D. McCalpain, "STREAM: Sustainable memory bandwidth in high performance computers," Tech. Rep., 1991–2007.

[24] B. Hadri *et al.*, "Overview of the KAUST's Cray X40 System – Shaheen II," in *Proceedings of the Cray User Group Meeting*, Chicago, USA, May 2015.

[25] B. Hadri, S. Kortas, R. Fiedler, and G. S. Markomanolis, "Regression testing on shaheen cray xc40: implementation and lessons learned," in *Preceedings of the Cray Users Group Meeting (CUG2017)*, 2017.

[26] K. Kandalla, P. Mendygral, N. Radcliffe, B. Cernohous, D. Knaak, K. McMahon, and M. Pagel, "Optimizing cray mpi and shmem software stacks for cray-xc supercomputers based on intel knl processors," in *Proceedings of Cray User Group 2016*, 2016.

Appendix

This appendix gathers the source codes along with the instructions for installation of the libraries dependencies, compilation and submitting jobs needed to reproduce the results shared for this paper. This paper is using NEKBOX and LIBXSMM, released as open sources.

A. Description

1) Overview:

- Algorithm: marching and SEM algorithms, Iterative solver, optimized small matrix matrix multiplication
- Program: FORTRAN, C, MPI, python
- Compilation: Intel Programming Environment, Intel 16,17,18 using the version of MPICH associated to the given CDT.
- Binary: nek5000
- Data set: output files containing timing and iterations
- Run-time environment: CLE 5.2 and CLE 6.0 on Shaheen (CDT 16.01, 17.12, 18.03, 19.03) and CLE6.0 on Trinity and Cori (CDT 16.10)
- Hardware: Cray XC40 using either dual sockets of Haswell 16 core Intel® Xeon® E5-2698v3 @ 2.3GHz on single-socket Intel® Xeon Phi Processor 7250 "Knights Landing") processor with 68 cores per node @ 1.4 GHz
- Run-time state: All runs are performed during production mode on Cori, Shaheen, except for full scale runs on Shaheen and Trinity
- Execution: Running on a maximum of 16 cores per socket on Haswell and disabling the hyperthreading. For the KNL runs, only 64 cores have been used for both cache and flat mode
- Output: Extract timing and efficiency from run logs
- Experiment workflow: Compile and run test cases
- Experiment customization: Depending on the KNL mode
- Publicly available?: Yes

2) How software can be obtained:

- NEKBOX: git clone https://github.com/NekBox/NekBox
- LIBXSMM: git clone https://github.com/hfp/libxsmm

3) Hardware dependencies: Code tested on x86 mainly Intel Haswell and Xeon Phi Knights Landing

4) Software dependencies: It is recommended to use anaconda for automatic compilation and building cases along with the fully functional nekpy

5) Datasets:

- Test cases: https://github.com/michel2323/GB16

B. Installation

```
wget https://repo.continuum.io/miniconda/
    Miniconda3-latest-Linux-x86_64.sh -O
    miniconda.sh
bash miniconda.sh -b -p $HOME/miniconda
export PATH=$HOME/miniconda/bin:$PATH
conda config --set always_yes yes --set
    changeps1 no
conda install numpy pytest chest cloudpickle
pip install graphviz mapcombine
pip install dask==0.9.0
pip install nekpy
```

1) Dependencies tools installation:

```
git clone https://github.com/hfp/libxsmm
module load cdt
cd libxsmm/
make
```

2) Installation steps of LIBXSMM on Haswell nodes on Cray XC40 :

```
module load fftw iobuf
module unload darshan

git clone https://github.com/NekBox/NekBox
git clone https://github.com/michel2323/GB16
git clone https://github.com/maxhutch/nek-
    tools/

export PATH=$HOME/NEKBOX/:$PATH
vi $HOME/NEKBOX/makenek
~/miniconda/bin/python3 ~/nekbox-tools/genrun
    /genrun.py -d eddy_uv.json -u eddy_uv_f90
    .tusr  test_case
```

3) Installation steps of NEKBOX on Haswell nodes Cray XC40: Optimization flags used are "-O3 -xCORE-AVX2 -fma -ip -ipo". Similar steps for the KNL compilations, except the swap of module craype-haswell to crape-mic-knl

C. Experiment workflow

Once NEKBOX is installed, the runs are then built as follows:

```
~/anaconda3/bin/python ./run_scaling.py
    cluster.json ../duct.tusr
```

In run_scaling script, you can adjust the array job_sizes where you can add the node numbers you want to run on. Then submit the jobs using run_all.sh and edit it with the proper workload manager. Finally, the script analyze.sh extracts the relevant data from the outputs.

D. Evaluation and expected result

The expected results is a log file similar to Nek5000, with a detailed information of time iterations steps variable values and reports of timing along with performance number.

E. Experiment customization

For runs on KNL, the cache and flat mode have been explored and in order to get the best results for the flat mode, the option numactl -p 1" needs to be added during the execution.