Research Paper

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2025, Vol. 39(3) 329–344

DOI: 10.1177/10943420241303163
journals.sagepub.com/home/hpc
Sage

Experience and analysis of scalable high-fidelity computational fluid dynamics on modular supercomputing architectures

Martin Karp[1], Estela Suarez[2,3], Jan H Meinke[2], Måns I Andersson[4], Philipp Schlatter[1,5], Stefano Markidis[4] and Niclas Jansson[6]

Abstract

The never-ending computational demand from simulations of turbulence makes computational fluid dynamics (CFD) a prime application use case for current and future exascale systems. High-order finite element methods, such as the spectral element method, have been gaining traction as they offer high performance on both multicore CPUs and modern GPU-based accelerators. In this work, we assess how high-fidelity CFD using the spectral element method can exploit the modular supercomputing architecture at scale through domain partitioning, where the computational domain is split between a Booster module powered by GPUs and a Cluster module with conventional CPU nodes. We investigate several different flow cases and computer systems based on the Modular Supercomputing Architecture (MSA). We observe that for our simulations, the communication overhead and load balancing issues incurred by incorporating different computing architectures are seldom worthwhile, especially when I/O is also considered, but when the simulation at hand requires more than the combined global memory on the GPUs, utilizing additional CPUs to increase the available memory can be fruitful. We support our results with a simple performance model to assess when running across modules might be beneficial. As MSA is becoming more widespread and efforts to increase system utilization are growing more important our results give insight into when and how a monolithic application can utilize and spread out to more than one module and obtain a faster time to solution.

Keywords

Computational fluid dynamics, modular supercomputing architecture, HPC

1. Introduction

Computational fluid dynamics (CFD) impacts many fields ranging from medicine to aeronautics and is one of the largest application domains in modern HPC systems (Slotnick et al., 2014). Designing efficient CFD software tailored to the most powerful supercomputers is an active area of research and developing methods and algorithms that map to upcoming heterogeneous hardware is growing ever more important (Abdelfattah et al., 2021).

The modular supercomputing architecture (MSA) is uniquely positioned as one of the main enabling technologies for the European exascale computer ecosystem. It combines different modules tailored for specific sets of algorithms and applications connected with a high-performance interconnect. This type of supercomputing cluster provides a dynamic and flexible system for a wide range of applications and use cases (Kreuzer et al., 2021; Suarez et al., 2019). It has already been deployed in both the JURECA and JUWELS supercomputers at Jülich

[1]FLOW, Engineering Mechanics, KTH Royal Institute of Technology, Stockholm, Sweden
[2]Jülich Supercomputing Centre, Institute for Advanced Simulations, Forschungszentrum Jülich GmbH, Jülich, Germany
[3]Institute of Computer Science, University of Bonn, Bonn, Germany
[4]Department of Computer Science, KTH Royal Institute of Technology, Stockholm, Sweden
[5]Institute of Fluid Mechanics (LSTM), Friedrich-Alexander Universität (FAU) Erlangen-Nürnberg, Erlangen, Germany
[6]PDC Centre for High Performance Computing, KTH Royal Institute of Technology, Stockholm, Sweden

Corresponding author:
Martin Karp, School of Engineering Sciences, KTH Royal Institute of Technology, Osquars Backe 18, 100 44 Stockholm, Sweden.
Email: makarp@kth.se

Supercomputing Centre (JSC) and is posed to be the computing architecture for a future exascale computer system at JSC (Krause 2019; Krause and Thörnig 2018). However, applications need to be adapted to take advantage of more than one module at a time.

Through dedicated efforts, MSA has already been accommodated in several applications such as multiphysics or multiscale applications that can efficiently run large well-defined code sections on different computing modules (Kreuzer et al., 2021; Markov et al., 2019; Riedel et al., 2021). By splitting the code execution and running large parallel regions on the Booster modules dedicated to energy-efficient high-throughput processing units such as GPUs and running portions with low scalability on the cluster module focused on providing low latency and high frequency, a large improvement in the performance of the solver has been observed (Kreuzer et al., 2018). However, some application domains are dominated by large homogeneous "monolithic" solvers where each process executes the same operations and only the computational domain is partitioned.

The benefits of MSA for these types of applications, which occur in various domains revolving around solving one large partial differential equation such as solid mechanics or fluid dynamics, are less clear-cut. On a large scale, when the problem will not fit on any one module, it comes down to distributing the work between different modules appropriately. For smaller problems, it instead becomes an issue of choosing the most suitable module to execute the computation on. As flexible job scheduling is becoming more important to increase system utilization, understanding the performance implications of using multiple modules for these types of applications is also becoming more relevant (Arima et al., 2022). One aspect of this work is to assess when utilizing several modules can reduce the time to solution for scalable monolithic solvers.

In this work, we evaluate how large-scale high-fidelity computational fluid dynamics simulations based on solving the Navier-Stokes equations can utilize different MSA modules at the same time and how workloads of different sizes are best run on a heterogeneous MSA system. High-fidelity CFD makes up a large share of the computational load on many supercomputers, and due to the demand for more grid points and higher resolution, there is a never-ending need for computational resources. This approach differs from lower-fidelity models such as the Reynolds-averaged Navier-Stokes or other approaches more suited for complex geometries such as Lattice-Boltzman, where the Boltzmann equations are solved instead. We use a CFD solver that performs well on both CPUs and GPUs combined with a simple performance model to analyze and understand how we distribute a workload and execute computations on two different MSA systems, the JUWELS cluster and Booster modules as well as the DEEP cluster and booster modules. We claim the following contributions:

- We empirically compare different flow configurations across different GPU/CPU configurations, utilizing not only GPUs and CPUs but also mixing the two architectures on MSA. We also evaluate the impact of I/O on the load balance.
- We employ a simple performance model to reason about our results and evaluate the performance potential by running on multiple architectures.
- When the simulation cannot fit on the GPU module only, by using both GPU and CPU modules, we observe up to 2.7× improved performance than only using the CPU module on the DEEP prototype system. We also compare the performance between the JUWELS Booster and LUMI-G module.

2. Related work

This work relates both to various applications utilizing multiple modules on MSA, as well as CFD in general on heterogeneous computer architectures. While most efforts for CFD have been spent on optimizing the code for systems where the nodes internally are heterogeneous, our work explores how a solver optimized for different types of nodes can run using multiple compute modules with different node architectures by partitioning the computational domain between the different modules.

2.1. CFD on heterogenous architectures

In the era of heterogeneous platforms, high-order methods for CFD have been gaining increasing amounts of interest for high-fidelity CFD due to their accuracy, structure, and relatively high number of floating point operations per grid point which enable them to efficiently utilize GPUs in addition to multicore CPUs (Abdelfattah et al., 2021). In the development of these methods, the focus has been on offloading the computation to the accelerator and limiting the data exchange from the host to the device as far as possible.

In this paper, to assess the performance of mixing different architectures, we consider a spectral element solver, Neko, running on nodes composed of CPUs as well as nodes powered primarily by GPUs with a host CPU. Neko uses modern Fortran together with hand-written CUDA/HIP kernels behind a device abstraction layer to provide tuned implementations for all the different architectures (Jansson et al., 2023, 2024; Karp et al., 2023). While there are many other methods to carry out fluid simulations, we focus on the Neko application, which integrates the Navier-Stokes equations in time and is able to efficiently scale using domain decomposition. CFD can take many forms on heterogeneous computer architectures, ranging from compressible solvers (Witherden et al., 2014) to Lattice-Boltzman methods (Calore et al., 2019) and many others

(Niemeyer and Sung, 2014). However, not all solvers scale to the same extent as Neko and can utilize different computer architectures at a high parallel efficiency. For our work on high-fidelity CFD running on large-scale heterogeneous architectures, the spectral element method (SEM) is a good representative, and two SEM codes were because of this recently considered for the Gordon-Ball prize (Jansson et al., 2023; Merzari et al., 2023).

There are many approaches targeting CFD, utilizing both CPUs and GPUs, as there are also different ways of utilizing mixed CPU-GPU nodes. Within a node, some approaches try to either offload certain tasks to the host CPU (Borrell et al., 2020; Calore et al., 2019), or partition the computational domain between computing devices depending on their respective performance (AlOnazi et al., 2015; Liu et al., 2016; Zhong et al., 2014). In our work, we are concerned with the second approach, but with the difference that we split the domain between two different computer modules. The works by Zhong et al. (2014); AlOnazi et al. (2015) indicate that partitioning the domain between different computing devices can lead to improved performance, but this is in practice not done in many large-scale CFD solvers (Abdelfattah et al., 2021; Kolev et al., 2021) because data movement between the CPU and GPU quickly becomes the limiting factor. Our work aims to assess why and when a CFD application should consider using a mixture of different computing modules, assuming optimal load balancing. We are the first, to our knowledge, to study the performance of a CFD code for large scale production runs on a mix of compute modules with hundreds of GPUs or thousands of cores. The motivation of this work is first to enable running large-scale monolithic solvers such as Neko across compute modules when the HPC cluster is underutilized, and second to determine from the application point of view when mixing modules is compelling for actual production cases.

2.2. Applications on MSA

Different applications have been tested on the MSA. In particular, large performance improvements have been made possible for applications employing coarse-grained parallelism, in which different parts of the code benefit from different computer architectures and only limited communication between the compute modules is necessary. Notable examples are the implicit particle in cell method in xPIC by Kreuzer et al. (2018) and machine learning Riedel et al. (2021). Further approaches across a wide range of applications reported in Kreuzer et al. (2021). However, as mentioned, the primary focus has been on dedicating specific computational resources to code parts with very different computational characteristics.

Our code, on the other hand, simulates an incompressible flow that lacks coarse-grained isolated tasks; instead, we partition the domain between different computing devices. Going forward we see an opportunity for workflows where several coarse-grained tasks are executed in parallel, in addition to the actual simulation. One such approach, where in situ data analysis is executed in parallel to the Neko simulation is suggested by Ju et al. (2023). While we focus on domain-partioning in this paper, considering such approaches, and for example running the in situ data analysis on a different module than the simulation is a natural extension to this work.

3. Computational fluid dynamics in HPC

Fluid dynamics has been one of the focus areas of high-performance computing since its conception. Due to the vast array of application areas such as medicine, aerodynamics, and weather and climate models, detailed simulations of flows are of large scientific interest. High-fidelity simulations of the turbulent Navier-Stokes equations require tremendous computing power and a very fine resolution making them prime candidates for taking advantage of large, modern HPC systems. In this work, we focus on the integration in time of the non-dimensional incompressible Navier-Stokes, described by

$$\begin{aligned} \nabla \cdot \mathbf{v} &= 0, \\ \frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v} &= -\nabla p + \frac{1}{Re}\nabla^2 \mathbf{v} + \mathbf{F}, \end{aligned} \tag{1}$$

where $\mathbf{v}$ is the instantaneous velocity field, p the pressure, Re is the non-dimensional Reynolds number and $\mathbf{F}$ an external forcing. The Reynolds number is defined as $Re = LU/\nu$ where U is a characteristic velocity, L is a suitable length scale, and ν is the kinematic viscosity. The Reynolds number is important in this context as a single direct numerical simulation of these equations, where all the scales of the flow are resolved, requires a grid that scales as $\mathcal{O}(Re^{9/4})$ for isotropic, homogeneous turbulence. This means that direct numerical simulation at even moderately high Reynolds numbers is extremely expensive. While there are many other approaches to CFD, our focus is on the integration of the Navier-Stokes equations in time with low numerical dispersion and high scalability. In our context, methods such as SEM are the prime candidates (Deville et al., 2002).

3.1. Neko

To assess how high-fidelity CFD simulations can be efficiently performed on varying computer hardware we will be utilizing Neko (Jansson et al., 2024), a Navier-Stokes solver based on the spectral element method. It has its roots in the long-running solver, Nek5000 (Fischer et al., 2008), which has scaled to over a million MPI ranks and was awarded the

Gordon Bell price in 1999 (Tufo and Fischer, 1999). Neko provides the same excellent scaling capabilities as Nek5000 on modern multicore systems and adds support for more recent computer architectures such as GPUs (Karp et al., 2023). This makes it a suitable candidate to assess how we can leverage a wide range of different computer architectures for large CFD simulations.

While several other methods are used for CFD, not all can utilize GPUs efficiently or scale to a large number of MPI ranks. Oftentimes a low operational intensity, the number of floating operations executed per byte, and the prevalence of complex global communication patterns make it difficult to utilize massively parallel architectures such as GPUs. Our choice of discretization and solver relates to this: the spectral element method has shown major promise in enabling CFD simulation at the exascale due to its high-order and local structure, enabling efficient utilization of both CPUs and GPUs (Abdelfattah et al., 2021; Fischer et al., 2020).

Due to the globally unstructured but locally structured nature of the spectral element method, only unit-depth communication is necessary in a so-called gather-scatter phase (Deville et al., 2002). All other operations can be performed in an element-by-element or matrix-free fashion, which yields a high level of parallelism and utilizes both multicore CPUs and GPUs efficiently. At the heart of the method, similar to many other CFD solvers, preconditioned Krylov subspace methods are used to solve linear systems on the form $Ax = b$ for each time step. The exact splitting of the velocity and pressure follows a similar splitting as outlined by Karniadakis et al. (1991) and described for Neko in Karp et al. (2023). For the resulting linear systems, we use restarted GMRES for the pressure solves with a hybrid-Schwarz multigrid preconditioner, while for the velocity we use CG together with a block-Jacobi preconditioner. While there are other pipelined Krylov methods and implementations available in Neko (Karp et al., 2022), for this study we evaluate the original and most common configuration.

In the spectral element method, the computational domain is split into E non-overlapping hexahedral elements. These parts of the domain are then distributed among the MPI ranks and it is through this domain partitioning that the spectral element method leverages the parallelism of modern computing architectures. The flow field is represented on the reference element with high-order polynomial basis functions of order N, collocated on the Gauss-Lobatto-Legendre points and is desrbied extensively in Deville et al. (2002). The computational load is identical for each element. The only asymmetry that is introduced is through the gather-scatter operation, which depends on the geometric distribution of the elements across the MPI ranks.

3.2. Flow cases under consideration

With our focus on high-fidelity simulations of turbulent flow, we consider three different simulation cases of varying sizes. We summarize the details of each flow case in Table 1. We use a polynomial order of $N = 7$ as most simulation cases use a polynomial order between 5 and 11.

3.2.1. Turbulent pipe. Turbulent flow in a pipe is a canonical flow case, which occurs in biological applications such as blood flow, and industrial applications such as gas and oil pipelines. One case that has been studied extensively, is the flow in a turbulent pipe at bulk Reynolds number $Re_b = 5300$ based on the cylinder diameter and bulk flow velocity U_b. We consider this case as a smaller simulation case, only requiring a few nodes to efficiently compute. The exact details of the flow case are described by El Khoury et al. (2013).

3.2.2. Taylor-green vortex. The Taylor-Green vortex (TGV) has been studied extensively in order to assess the accuracy and convergence of CFD solvers. In the TGV case, the Reynolds number is uniquely defined by the viscosity and in particular, TGV at $Re = 1600$ has been used previously (Van Rees et al., 2011). We use this case to assess the scaling behavior of a medium-sized workload requiring a moderate number of nodes to execute efficiently.

3.2.3. Rayleigh-Bénard convection. For our largest case we consider Rayleigh-Bénard convection (RBC), which simulates the same physical behavior that occurs in the sun and many industrial applications (Iyer et al., 2020) where the increased buoyancy of a hotter fluid drives convective turbulence as shown in Figure 1. In this work, we consider a cubic domain with an aspect ratio of 1, periodic sides, and walls on the top and bottom where the bottom wall has a temperature of 1. We perform this simulation at a Rayleigh number of 10^{11} and a Prandtl number of 1. Our simulation follows a similar setup to the cubic case in Kooij et al. (2018), but at a higher Rayleigh number. As this case is rather large we want to consider how to utilize several modules when one module might be too small to fit the entire problem.

Table 1. Flow cases under consideration. Polynomial order N, number of elements E, and total number of unique grid points, n.

Case	N	E	$n = EN^3$
Turbulent pipe $Re_b = 5300$	7	36480	12512640
Taylor-green vortex $Re = 1600$	7	262144	89915392
Rayleigh-Bénard convection, $Ra = 10^{11}$	7	2097152	719323136

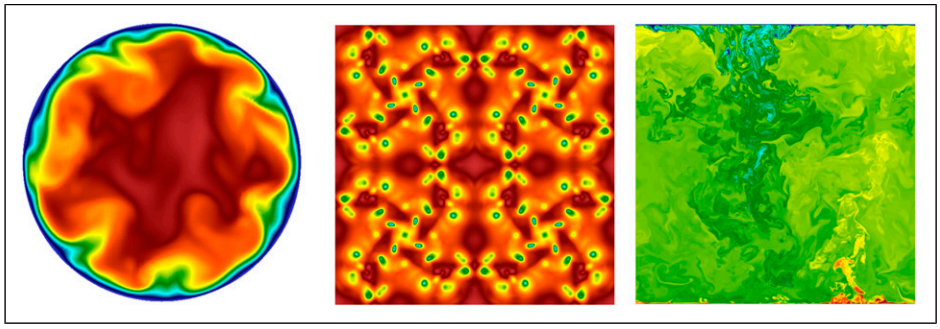

Figure 1. Visualizations of the three different cases, with red being high and blue being a lower value. To the left is the velocity magnitude in a cross-section of the pipe, in the middle is the pressure field in TGV and to the right, we show the temperature field in turbulent Rayleigh-Bénard convection.

4. Performance analysis

In this section, we perform a performance analysis where we relate the performance and memory capacity of different computing devices to reason around when and how it might be beneficial to split a homogeneous problem, where each device performs the same task on different parts of the problem, across different computing devices and supercomputer modules. We first develop a simple model to reason around the performance of mixing different computing devices and then go on to identify different domains of operation for a homogeneous workload, in what domains our performance model will work well, and what performance improvements one can expect in the best case by using different compute modules.

4.1. Performance model for mixing different computing devices

We develop a simple performance model for computations revolving around solving one large system by splitting a homogeneous computational cost (such as the computational domain) between different computing units (such as GPUs and CPUs). The aim of this model is to provide an optimistic indication of when using several computing modules might be beneficial, not to predict the exact run time of an application. The model is similar to what was originally proposed by Amdahl and similar to what has been used previously to discuss the performance and scalability of PDE solvers (Fischer, 2015).

We denote the execution time of a simulation with T and divide it into two non-overlapping sections:

$$T = T_a + T_c, \tag{2}$$

where T_a is the local time dedicated to arithmetic operations and loads and stores to and from global memory (DRAM or high bandwidth memory (HBM)), while the communication time T_c is the latency portion of the run time that is used for communication between different MPI ranks and inherent latency of the computing devices. We also introduce the computational cost or work C for a given workload which is then divided among all computing devices $C = \sum_{s_i \in S} C_i$. Each computing device $s_i \in S$, where S is the set of computing devices, then has a performance $P(s_i, C_i)$ given that computing device s_i is computing a cost of C_i. The units for C, P, depend on the problem, but in our case the cost is related to the computation of one time step, meaning that the cost is given in time steps and the performance in time steps per second. For a given processing device s_i computing a cost C_i we have that

$$T_a(s_i, C_i) = \frac{C_i}{P(s_i, C_i)}. \tag{3}$$

What we would like to obtain is the minimal run time overall computing devices, and hence solve the minimization problem

$$\begin{aligned} &\underset{C_i}{\text{minimize}}\ T = \max_{s_i \in S}(T_a(s_i, C_i) + T_c(s_i)) \\ &\text{such that. } T_a(s_i, C_i) = \frac{C_i}{P(s_i, C_i)} \\ &C = \sum_{s_i \in S} C_i \\ &C_i \leq C_{max}(s_i), \quad s_i \in S \end{aligned} \tag{4}$$

where we introduce the capacity of computing device s_i as $C_{max}(s_i)$, which is the largest cost a given computing device can compute, often limited by for example DRAM or HBM memory capacity. For our model, we focus on finding a lower bound on the run time and comparing the results of our performance measurements to this optimistic lower bound. To do this, we start by observing that $T \geq T_a$ and as such we can trivially lower bound the performance and run time for the computing device as

$$T \geq \frac{C_i}{P(s_i, C_i)} \geq \frac{C_i}{P_{opt}(s_i)} \tag{5}$$

where we introduce $P_{opt}(s_i)$, which corresponds to the highest performance achievable for processing device s_i.

With this information, we can provide a lower bound on the lowest possible run time T_{min} as

$$T_{min} \geq \max_{s_i \in S} \frac{C_i}{P_{opt}(s_i)} \tag{6}$$

subject to the constraint that $C = \sum_{s_i \in S} C_i$. For the unconstrained case, when all computing devices have enough memory to fit their part of the cost C, this reduces to

$$T_{min} \geq \frac{C}{\sum_{s_i \in S} P_{opt}(s_i)} \tag{7}$$

and the relation $C_i/P_{opt}(s_i) = C_j/P_{opt}(s_j)$, $\forall s_i, s_j \in S$ holds. In the other case, we have that there exists some computing devices s.t.

$$C_{max}(s_i)/P_{opt}(s_i) < C_j/P_{opt}(s_j), s_i, s_j \in S$$

and the optimization problem does not necessarily have a simple solution. As we consider only two different computing devices in this work (one kind of GPUs and CPUs used at the same time), solving this problem is not an issue, but if the performance $P(s_i)$ would vary significantly among the computing devices $s_i \in S$, the number of constraints would increase considerably. To summarize, our modeled lowest possible run time of our mixed GPU/CPU runs is computed as the following:

$$\begin{aligned}
&\text{If } \frac{C_i}{P_{opt}(s_i)} = \frac{C_j}{P_{opt}(s_j)}, \quad \forall s_i, s_j \in S, \\
&\quad C = \sum_{s_i \in S} C_i, \\
&\quad C_i \leq C_{max}(s_i), \quad s_i \in S, \\
&\text{then: } T_{min} = \frac{C}{\sum_{s_i \in S} P_{opt}(s_i)} \\
&\text{Else:} \\
&\underset{C_i}{\text{minimize}}\, T_{min} = \max_{s_i \in S} \frac{C_i}{P_{opt}(s_i)} \\
&\text{such that. } C = \sum_{s_i \in S} C_i \\
&\quad C_i \leq C_{max}(s_i), \quad s_i \in S
\end{aligned} \tag{8}$$

The best case is that the performance of two computing devices is additive if they can fit the entire problem. Another takeaway from this model is that we can achieve significant superlinear speedup when a single module of computing devices cannot hold the entire computational cost and we are limited by the capacity of the devices. Increasing the capacity then effectively yields a superlinear speedup until the modules can hold enough of the computational work. We illustrate the meaning of our notation in Figure 2, for a simple case with two different computing devices, s_1, s_2. Given the single node performance shown in Figure 2(a), the modeled performance as we scale is shown in Figure 2(b).

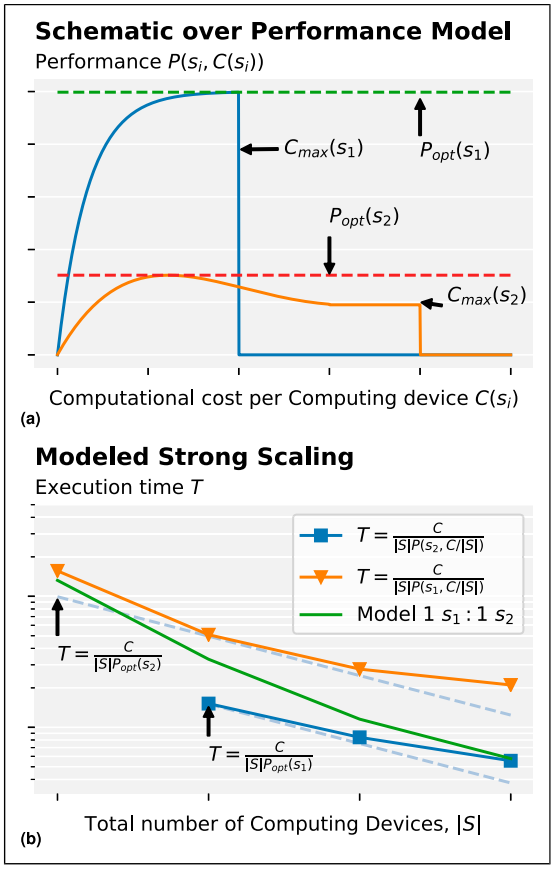

Figure 2. Illustration of the performance model for two different computing devices s_1, s_2 with different performance characteristics. We denote the modeled time as described in (8) as Model 1 s_1: 1 s_2 and we model the best achievable performance based on $P_{opt}(s_1)$, $P_{opt}(s_2)$ with a mix of 1:1, s_1, s_2 devices. The strong scaling performance for $|S|$ computing devices with a performance based on Figure 2(a) is shown as in Figure 2(b).

For Neko, we let the cost C be a linear function of the number of elements, E_i, on a computing device and model the performance according to equation (8). As such, finding T_{min} can be done through a parameter search where we load balance the elements between the different computing devices. The best performance $P_{opt}(s_i)$ for the GPUs and CPUs is approximated as the best-measured performance for a given flow case, using only CPUs/GPUs. We visualize the modeled time with a solid line in our experimental results along with our mixed GPU/CPU runs, similar to the modeled strong scaling in Figure 2. A similar approach can be applied to any other solver solving one large problem through domain partitioning.

4.2. Operation domains

An aspect of the modeled time that we propose is that we do not consider the communication time T_c, but we assume that

the whole problem scales perfectly. This is most often not the case, but it depends strongly on the problem size, and thus the cost per computing device C_i which relates to the relation between T_a and T_c. As such, we introduce three different domains of operation for an application with different performance characteristics, where the computation is either dominated by T_a or T_c, and discuss where running on a mix of computing devices might be beneficial.

$$\begin{aligned} &T_a \leq T_c, && \text{Communication domain} \\ &T_a > T_c, && \text{Scaling domain} \\ &T_a \gg T_c, \quad C \approx C_{max}, && \text{Extreme scaling domain.} \end{aligned} \tag{9}$$

In the communication domain, it does not make sense to add computational resources, as T_c in general increases with the number of processing devices, and we are already limited by communication (latency). In this domain, CPUs may have an edge due to their low latency concerning memory and communication and high clock speeds. This is the case for many applications, which do not have the opportunity to scale on GPUs or to a large number of nodes and this is the domain the Cluster module caters to.

In the scaling domain, the total amount of work is still the dominating factor for the application performance, hence adding more compute units would be beneficial. However, in this domain, it is still not evident that we will easily be able to balance the different computing units in such a way that we get a reduction in run time. However, as we are primarily limited by computational power, throughput-oriented devices such as GPUs tend to be the most performant and power-efficient option, which is the idea behind the Booster module (Kreuzer et al., 2021).

In the extreme scale regime, we are considering examples in which the computational cost C is close to the capacity $C_{\max}$ of the available resources and might not fit into any single compute module. In this situation, the ability to use several modules to fit a large case becomes crucial, which justifies the potential loss in workload balance. The total performance, assuming T_c is small, will overall be additive and follow our performance model, and the major appeal is that cases that are impossible to run otherwise will now be possible. Overall, these cases would then not treat the Cluster and Booster modules as two different modules, but rather as two pillars to compute these extremely large systems. This domain most closely correlates with our proposed performance model, while the model would provide optimistic performance bounds in the first two domains.

Neko, similarly to many flow solvers is primarily memory bound for the computational cost C, while the communication overhead, T_c, can be primarily attributed to the gather-scatter kernel. This is consistent with previous works where the gather-scatter kernel has shown to be the main performance bottleneck of SEM as one approaches the strong scaling limit, and a heavily optimized version is integral for high performance (Ivanov et al., 2015). The gather-scatter kernel is called repeatedly for each operator evaluation and has a strong dependence on the distribution of the work among the available ranks as it performs the unstructured communication among MPI ranks and elements.

5. Experimental setup

In this work, our primary experimental platforms are based on the modular supercomputing architecture (MSA) (Suarez et al., 2019). MSA groups different kinds of compute nodes into sub-clusters (modules) that are internally rather homogeneous. The node architecture of each module targets the needs of a specific kind of application. Depending on the required network topology new modules can be added and extended easily.

An example is the JUWELS supercomputer—one of the largest systems in Europe—at the Jülich Supercomputing Center. It currently accommodates two different computing modules (Cluster and Booster) that share a single high-performance interconnect. With this design, it is possible to dynamically map applications with vastly different performance characteristics to the modules and accommodate a wide range of use cases. The JUWELS Cluster is a CPU-based HPC system, good for applications (or parts of them) that are not ready to run on GPUs and/or require high single-thread performance. The Booster module utilizes GPUs and is used by the most scalable applications with high-performance demands.

The DEEP system, a prototype for the modular supercomputing architecture provides in addition to a cluster and a booster module, a module dedicated to data analytics. This module is equipped with large, fast, storage as well as GPUs and FPGAs for extensive data processing. By sharing the same interconnect it is possible to assign different tasks to the modules that are executed in situ while the simulation is running.

Aside from the two systems just described, we evaluate the LUMI supercomputer at CSC in Finland. While LUMI shares a similar modular architecture to the systems at JSC, with different modules for CPU and GPUs, the vast amount of the resources is dedicated to the GPU/Booster module LUMI-G, which we will consider. We focus on the three production use cases described in subsection to capture actual production usage and do not evaluate any proxy app or similar, but the whole application. For all measurements we use a shaded area to indicate the 95% confidence interval for the time of any time step of the simulation, assuming that the time per time step follows a normal distribution around the sample mean. We use the last 100 time steps of each simulation to collect our performance measurements.

We provide an overview of the different computational setups and the two modules of each that we use in Table 2. A major difference from LUMI-G as compared to the Booster module of JUWELS is that the network interface cards (NIC)

Table 2. Software details and hardware details per node of the different computer modules and setups.

JUWELS	Cluster	Booster
Compute nodes	2271	936
CPU	2 × 24 core Intel Xeon 8168	2 × 24 core AMD EPYC 7402
CPU memory	96 GB DDR4-2666 RAM	512 GB DDR4-3200 RAM
GPU	-	4x Nvidia A100
GPU memory	-	40 GB HBM
Interconnect	Mellanox InfiniBand EDR100	4 × Mellanox HDR200 InfiniBand
Compiler	GCC 11.3.0	GCC 11.3.0
CUDA/ROCM	-	CUDA 11.7
MPI	OpenMPI 4.1.4	OpenMPI 4.1.4
DEEP	**Cluster**	**Booster**
Compute nodes	50	75
CPU	2 × 12 core Intel Xeon 6146	2 × 8 core Intel Xeon 4215
CPU memory	192 GB DDR4	48 GB DDR4
GPU	-	Nvidia V100
GPU memory	-	32 GB HBM
Interconnect	Mellanox InfiniBand EDR100	Mellanox InfiniBand EDR100
Compiler	Intel 2021.4.0	Intel 2021.4.0
CUDA/ROCM	-	CUDA 11.7
MPI	ParaStationMPI 5.5.0	ParaStationMPI 5.5.0
LUMI		**LUMI-G**
Compute nodes		2560
CPU		1 × 64 core AMD EPYC 7A53
CPU memory		512 GB DDR4
GPU		4 × AMD Instinct MI250X
GPU memory		128 GB HBM2e
Interconnect		4 × 200 GB/s Slingshot-11
Compiler		CCE 14.0.2
CUDA/ROCM		ROCM 5.0.2
MPI		Cray-mpich 8.1.18

are mounted directly on the GPUs, essentially offloading also the communication in addition to the computation to the GPU. On JUWELS, in comparison, the Mellanox HDR200 is connected to the GPUs through a PCIe switch that is shared with the host CPUs. The topology of the networks in the computers also differs: LUMI-G is arranged in a more conventional Dragonfly topology (Kim et al., 2008), while JUWELS uses a Dragonfly + network topology as proposed by Shpiner et al. (2017). All runs in Neko are executed with one MPI rank per CPU core for the CPU nodes, and one MPI rank per logical GPU for the GPU nodes. For our experiments mixing GPUs and CPUs, we use Neko extended with support to distribute the number of elements unevenly between different MPI ranks. For the distribution of the elements we then first partitioned the mesh with ParMETIS Karypis et al. (2003) and after this, we performed a parameter search to find the best weight (how many elements each core/GPU should compute) between the GPU and CPU devices for each case.

For the execution of the inter-module cases, we utilized the heterogeneous job scheduling available on JUWELS and DEEP. As we are comparing a wide range of computational platforms we introduce the notion of a computing device for a computational platform. We define the computing devices for each platform as one CPU node on DEEP/JUWELS or one logical GPU, meaning one Graphics Compute Die (GCD) of the MI250X or one V100/A100 GPU. We provide an overview in Table 3. In our mixed runs, we use a mixture of one CPU computing device and one GPU computing device to illustrate the performance behavior when mixing computer modules.

We utilize LLView on the DEEP system to collect statistics for MSA runs with and without significant amounts of I/O. This is to identify how the workload and load balance changes if the I/O load increases compared to the computational workload.

Table 3. List of computing devices used in the experiments. For the MSA runs we utilize a 1:1 Mix where one computing device from the Booster and Cluster module is used simultaneously.

Compute cluster	Computing device
DEEP-booster	1 V100 GPU
DEEP-cluster	1 node with 2 × 12 Intel CPU cores
JUWELS-booster	1 A100 GPU
JUWELS-cluster	1 node with 2 × 24 Intel CPU cores
LUMI-G	1 MI250X GCD

6. Results

In this section, we detail the performance measurements for the different simulation cases across the experimental platforms and discuss how the results relate to our previous performance analysis. We show the standard deviation with a shaded area in all plots.

6.1. Performance measurements

We have collected the majority of the runs and comparison between DEEP and JUWELS into Figure 3 together with the modeled best-case performance for the MSA runs. We see that the GPUs significantly outperform the CPUs for Neko, similar to (Karp et al., 2023), while the strong scaling behavior when using GPUs is significantly worse. Scaling on the CPU clusters is nearly linear with a parallel efficiency between 90% and 110% in almost all cases. The superlinear speedup we observe in for example the Pipe and TGV case on JUWELS is a well-known property of the spectral element method when strong scaling on multicore CPUs, this is discussed in for example Offermans et al. (2016). For the GPUs, we achieve a parallel efficiency of 80% for the first points while it decreases towards 50%–60% when we have 4000 or fewer elements per GPU. We observed that in general, it was beneficial to put as many elements as possible on the GPU when in the extreme scaling domain, where the computing devices are close to their max capacity, due to their high performance. When in the scaling domain however, putting more elements on each CPU core gave the best performance, with each GPU computing around $60 - 120$ times the number of elements compared to a single CPU core.

Focusing first on the turbulent pipe case shown at the top of Figure 3 we see how the performance is affected by distributing the computation between different computer architectures. As is proposed in the performance model, the performance is between the GPU and CPU performance and aligns well with the modeled line for the DEEP cluster. However, it becomes evident that the performance of this use case on JUWELS does not benefit from MSA, as the problem is small enough to be efficiently run on a single GPU node with four GPUs and 9000 elements per GPU. The performance model for the best possible execution time follows a perfect linear scaling from one GPU. Therefore, it provides a very optimistic bound for JUWELS, significantly overpredicting the performance, because it does not take into account the impact of network communication when scaling beyond one GPU node. We also performed measurements using the local CPUs of the GPU nodes, but even in this case, the communication overhead surpassed the potential performance gain from using more CPUs on JUWELS. This can be partially explained by the vast imbalance between the GPU and CPU nodes on JUWELS, where the DDR memory of the host CPUs offers less than 10% of the accumulated memory bandwidth of HBM memory on the GPUs. Partitioning the domain then leads to expensive memory transfers over PCIe.

The primary case for MSA here would be when only one GPU is available, which cannot fit the entire problem. This is the case on the DEEP system. Using 1 GPU and 1 CPU node on DEEP results in more than 2× speedup compared to 2 CPU nodes. Using both the GPU and CPU could thus potentially be beneficial for personal computers and desktops where the global memory of the GPU can not accommodate the entire problem. Of note is that the imbalance is lower on DEEP, as the number of GPUs per node is smaller. Using additional CPUs, both on the same node or another module yields here a proportionally larger performance improvement.

For the TGV case, we see a similar performance curve to that of the turbulent pipe where the GPU and CPUs perform similarly. As for the MSA runs, we see that the performance for a few nodes is rather low as the CPUs need to carry a vast amount of memory, we are in other words limited by the $C_{\max}$ of the GPUs, meaning that the CPUs must carry out the majority of the computational cost. For 16 computing devices, however, we find ourselves in the domain of our model where we can obtain additive performance in the best case as we scale up. For DEEP we get within 10%–15% of the best possible time for 8 and 16 computing devices, while we are within 10% for eight computing devices on JUWELS. For 32 on DEEP and 32–64 on JUWELS the communication time T_c quickly impacts the performance we can achieve and the actual performance deviates more than 20% from the modeled best-case, but the curve starts to align with the GPU-only scaling. We observe that for all cases up to 32 devices, the modeled performance predicts a worse performance than using the same number of GPU computing devices. For 64 devices the modeled performance of the MSA run would perform equally to the measured performance of 64 GPUs, assuming perfect scaling. At this point, however, the internal latency (T_c) of the computing units and communication overhead is significant, leading to a worse performance than modeled.

For the largest case, RBC, our results differ in some regards from the previous cases. As the Rayleigh-Bénard case has more than 2M elements, we cannot fit the problem on the DEEP Booster module where the GPUs only have 32 GB of HBM memory per GPU. We want to compare the number of computing devices between Cluster and Booster

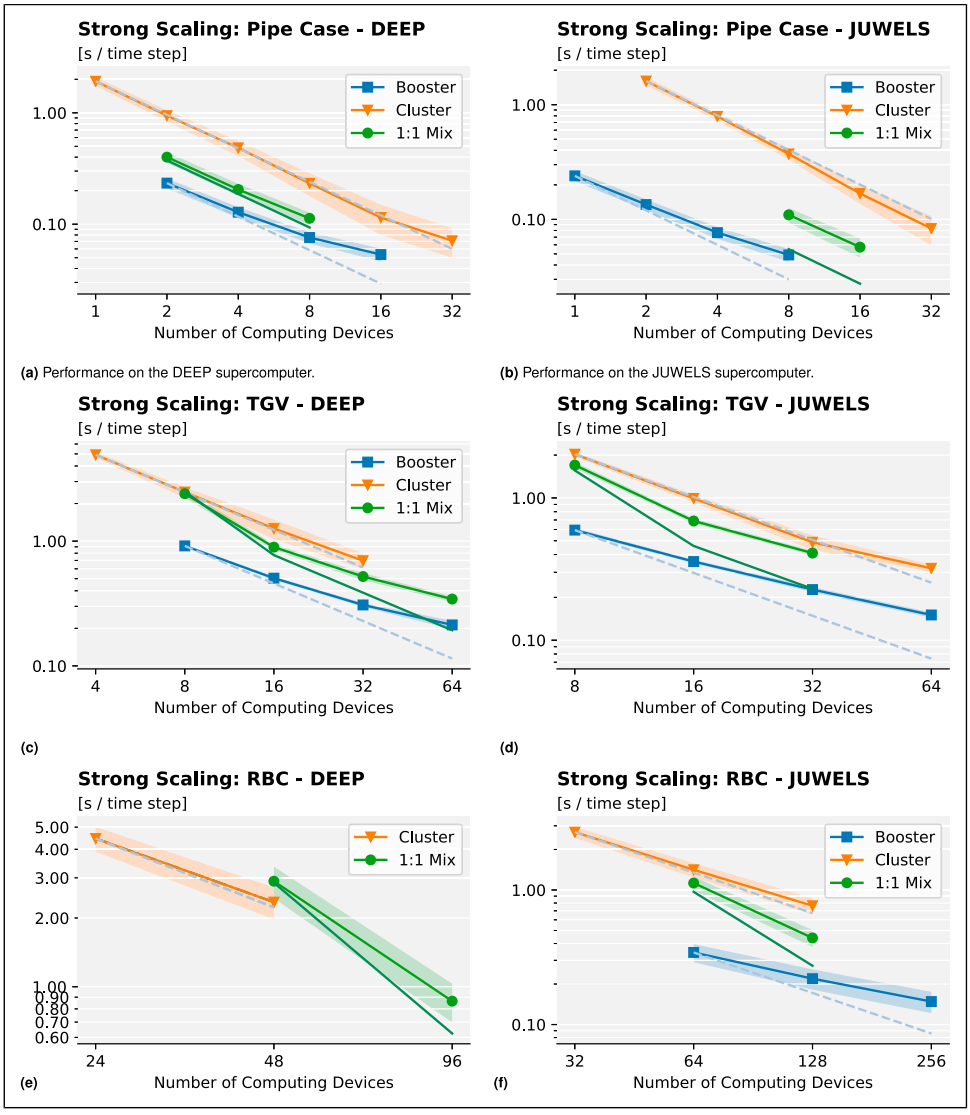

Figure 3. Performance comparison between DEEP and JUWELS for our three different test cases. We show perfect linear scaling for the Booster and Cluster runs with a dotted line while we show the modeled performance with a green solid line without markers for the MSA runs. The modeled time is based on the highest performance P_{opt} for the given case measured on the Booster and Cluster modules. (a), (c), and (e) show performance on the DEEP supercomputer. (b) (d) and (f) show performance on the JUWELS supercomputer.

fairly, which prevented us from computing the problem with 48 GPUs, because the memory requirement is around 1 GB of memory per 1000 elements (meaning a total memory requirement of 2000 GB for the RBC case) for polynomial order 7. As such we perform measurements only on the Cluster, comparing to the use of both the Booster and Cluster modules. The modeled best case then is based only on the best CPU performance and the computational cost dedicated to the CPUs. Here we can clearly see the opportunity of running a modular job to enable large problems to be efficiently executed. By using 48 GPUs in addition to 48 Cluster nodes, and using almost the whole DEEP system, we obtain a speedup of 2.7× compared to using almost the whole CPU module. However, one should note that the performance actually decreases compared to the Cluster-only runs when we execute the computation with 48 computing devices on DEEP. This is because of the lower memory capacity of the GPU nodes, which means that the number of elements per core is larger than when using only 48 CPUs. For 48 devices the number of elements increases from 1820 to 2400 as each V100 GPU can only accommodate a bit more than 30,000 elements in the HBM memory. The cost C per core then grows, and the runtime also increases, as predicted by our model. As such, one needs to consider that replacing one module with a high memory capacity by one with a higher performance and

lower memory capacity does still decrease the cost per rank. Otherwise, the benefit of using a more powerful module does not improve the performance. This is no longer the case when using 48 GPUs: they then have a large enough capacity to also decrease the work per core for the CPUs.

On JUWELS however, the performance increase is only prevalent for 64 computing devices, while using 64 GPUs + 64 CPU nodes gives a lower performance than only using 64 GPUs. As such the primary benefit of inter-module jobs for CFD applications is in the domain when T_c is comparably small and the Booster module does not have enough memory available to accommodate the problem. This corresponds to the extreme scale operation domain, which for Neko corresponds to more than 20,000 elements (for polynomial order 7), using half or more of the available HBM memory on the GPUs. It is only in this domain when additional computational resources are not as heavily affected by the different performance characteristics of the different modules and the performance is close to additive.

We also provide a comparison between the LUMI and JUWELS Booster modules for the RBC case in Figure 4. As we see in our measurements, CFD which can utilize GPUs is executed most efficiently on a large Booster-like system, we also provide this comparison between two current pre-exascale European supercomputers incorporating a modular design. As the best case in our measurements is to use the GPUs only to as large an extent as possible, we also include measurements with device-aware MPI enabled, where the MPI calls can be issued using pointers to memory on the device directly, further eliminating the host. One thing that is clear from the comparison between LUMI and JUWELS is that not using device-aware MPI on LUMI gives a significant performance penalty of 30%–50%, likely because the NIC is attached to the GPU using MPI on the host leads to unnecessary data movement. For JUWELS, we observe a negligible difference between using device-aware MPI and host MPI, and it performs similarly to using host MPI on LUMI. Overall, one A100 performs better than one GCD of the MI250X when the number of nodes is small, but when the number of nodes is increased the improved network on LUMI makes up the difference. The difference between device-aware MPI and host MPI on JUWELS is smaller than 5% and well within the standard deviation of a time step. This is in contrast with previous runs we executed using a mesh that was not load-balanced when device-MPI could perform as much as 6× better than using host MPI. These measurements indicate that if the problem is well partitioned, using device instead of host MPI does not make a big difference on JUWELS, but for ill-partitioned problems, the importance of device-aware MPI grows. On LUMI, not using device-aware MPI always gives a significant performance penalty, performing 30%–50% worse than with device-aware MPI enabled. Compared to the CPU only runs on JUWELS, we observed that CPUs were much

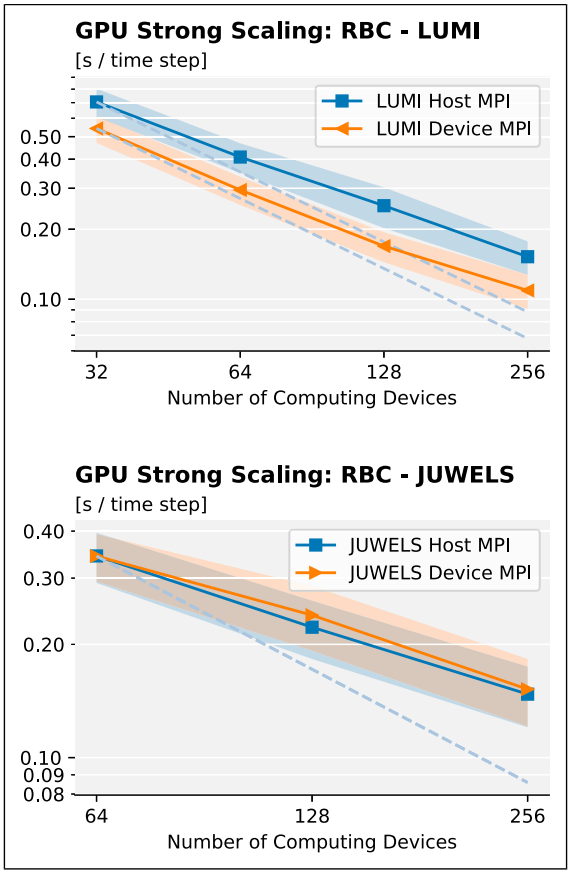

Figure 4. Performance comparison between the LUMI-G and JUWELS-Booster module where we compare utilizing the host for communication (host MPI) and utilizing device-aware MPI where the host is only used to schedule kernels on the device.

less affected by the partitioning of the elements between the different ranks. The large differences can be partially explained if we consider the node configuration on JUWELS Booster and LUMI-G and how the NICs are installed. For LUMI, they are attached directly via PCI Gen 4 to the GPUs whereas the host is not directly connected to the network as the node architecture is tailored for application where most data resides in the HBM memory of the GPUs (Atchley et al., 2023). This means that when device-aware MPI is not used, the data cannot be sent directly from the host, but it is first transferred from the GPU to the host CPU and then passed back through the GPU again before being communicated through the network. The same process is also applied when receiving messages. LUMI is thus not well suited for host-MPI. On JUWELS however, one would not expect the difference to be as pronounced since the PCIe switch is shared between the host and the GPU, and the data must not pass through the GPU an extra time when sending and receiving a message. However, still the difference compared to device-aware MPI is smaller than expected as one still executes two extra memory transfers to and from the

CPU for each message and does not utilize the direct communication between GPUs that device-aware MPI enables. A probable explanation of the limited impact observed for JUWELS is that the configuration of MPI we employ on JUWELS is not highly optimized for device-aware MPI. There is a significant number of options for the MPI runtime on JUWELS, for example, by configuring UCX we would be able to achieve better use of device-aware MPI at this scale. In particular, during the runs on JUWELS the unreliable datagram (UD) setting with CUDA transport for UCX was used, intended for medium-sized simulations. It is possible that the low-memory DC (Dynamically Connected) option might be more performant at this scale. This option, however, had at the time of carrying out these experiments not been exhaustively tested on the JUWELS system.

6.2. Modeled performance

In our performance model, we are interested in modeling the best possible execution time given a set S of computing devices. While we observed that it in some cases significantly overpredicts the performance for a mixed CPU-GPU run, it clearly illustrates how using only the strongest computing device to as large an extent as possible (assuming there are enough of them to accommodate the problem) is the way forward for large-scale homogeneous simulations. Although we have focused on CFD in our work, the same reasoning can be applied to any homogeneous workload where the main issue is to load balance parts of the problem between different ranks. As many applications fall in this category, our results support the trend of recent massively parallel systems to utilize primarily GPUs for the computation and dedicate a less powerful host only to schedule the computations. The latest candidates in this regard, LUMI, and Frontier, illustrate this trend clearly as the bandwidth and flop/s of the accelerators are more than 20× the performance of the host on a compute node. With upcoming architectures, we anticipate that the trend to remove the host from the computation and offload all tasks to the accelerator will continue. This is also the idea behind the Booster module where low-powered CPUs are equipped with powerful accelerators (Kreuzer et al., 2018). With this, we stress the point that for problems like CFD using a mix of CPU/GPU resources will likely not lead to any gains in the future, except for the case when the best-suited computing unit (in our case GPUs) cannot accommodate the entire problem. However, an opportunity is also to use applications such as this to backfill the computer resources when the system is idle. It is expected that incorporating more technologies such as malleable job-scheduling where jobs grow and shrink, applications that operate in the extreme-scale domain could use a simple performance model to indicate whether adding resources can be beneficial to decrease their time to solution. For Neko, in this case, our results indicate that the application operates in the extreme scale domain when more than half of the available memory is used on the GPUs. In a scenario where only some CPU resources are available directly to start the initialization of the problem, as more GPU resources become available the application accommodates more GPUs until the point at which the problem fits on only the Booster.

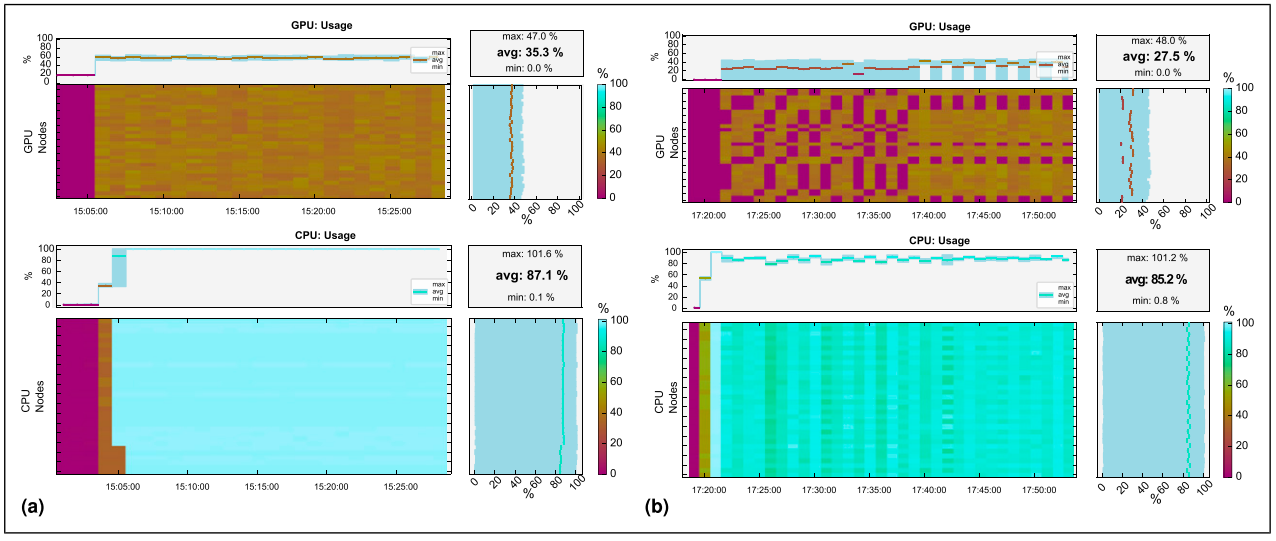

Figure 5. Performance traces with low-performance overhead from LLView for the GPU nodes (top) and CPU nodes (bottom) for an MSA run of the TGV case using 64 nodes split equally between GPU and CPU nodes (1:1 mix). The metric CPU and GPU usage are defined as the percent of time over the past sample period during which one or more kernels were executing on the GPU. (a) A trace with no I/O. (b) A simulation with extensive I/O is presented.

6.3. I/O and mixing modules

In the previous sections, we primarily considered the issue of balancing the load between different modules to the actual computation, however, for several applications I/O is the primary performance bottleneck. The impact of executing with a significant portion of I/O where output is written at each time step, versus one without any I/O is shown from LLView in Figure 5. From this, it is clear that not only must one then balance the computational load between the different computing devices, but also the writes to and from disk. The issue of balancing the load between devices can in the extreme case lead to a conflict between the computational load balance and the load on the file system. The I/O imbalance is due to the GPUs computing 100 times the number of elements compared to one CPU core, as such, on DEEP, this leads to the GPU nodes performing $100/24 \approx 4$ times more I/O, greatly impacting the GPU usage. This I/O imbalance leads to the GPUs spending a significant time idle compared to when not a lot of I/O is executed. The overall GPU utilization in this example is rather low though, as it is measured for the TGV case with 32 GPUs and 32 CPUs, and the problem size per computing device is comparably low.

7. Conclusions

Our results support the notion that if the numerical method can both utilize CPUs and GPUs efficiently, executing large-scale CFD on a Booster-like system is beneficial when the problem fits on only this module. There is some room for improvement in the use of a mix of CPU and GPU nodes when the problem size is too large for the GPU module and when the HBM memory of the GPUs cannot fit the entire computational load, for our Neko setting this requirements was 1 GB of global memory per 1000 elements, but this may vary between cases and for other applications. Overall, we observed that for this type of code where we utilize domain partitioning between the modules, the communication overhead quickly becomes larger than the potential gain from using multiple computing devices. This is further amplified when a significant amount of I/O is carried out. As the GPUs have a higher performance and carry out a larger amount of work, they also write significantly more data to the parallel file system. While the performance of the GPUs is significantly higher, the bandwidth to disk is comparable to the CPU nodes, leading to a significant imbalance. When the problem can fit on the GPUs only, it is best to utilize only the Booster, and even using the local host CPU gives a negligible or negative impact on the performance. For the GPU-only runs, we observe a difference between the JUWELS Booster and LUMI supercomputer when using device-aware MPI, primarily attributed to their respective network, and in particular to the NICs on LUMI being connected directly to the GPUs. The performance of one Nvidia A100 on JUWELS is higher than LUMI's AMD MI250X GCD for a few nodes, but using device MPI improves the scaling on LUMI. We observe that the trend of moving to larger GPU-accelerated systems, where not only computation but also communication is offloaded to the most powerful computing units to increase locality, will benefit computational fluid dynamics applications able to efficiently offload the whole algorithm to the accelerator.

Acknowledgements

The authors gratefully acknowledge the computing time provided by the Jülich Supercomputing Centre (on JUWELS and DEEP). We acknowledge the National Academic Infrastructure for Supercomputing in Sweden (NAISS) and the Swedish National Infrastructure for Computing (SNIC) for awarding this project access to the LUMI supercomputer, owned by the EuroHPC-JU, hosted by CSC (Finland) and the LUMI consortium through a LUMI Sweden XLarge call.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the European Union Horizon 2020 research and innovation programme under grant agreement No 955606 (DEEP-SEA). The EuroHPC Joint Undertaking (JU) receives support from the European Union Horizon 2020 research and innovation programme and Germany, France, Spain, Greece, Belgium, Sweden, Switzerland. Financial support was provided by the Swedish e-Science Research Centre Exascale Simulation Software Initiative (SESSI) and the Swedish Research Council project grant "Efficient Algorithms for Exascale Computational Fluid Dynamics" Vetenskapsrådet, (grant reference 2019-04723).

ORCID iDs

Martin Karp https://orcid.org/0000-0003-3374-8093
Philipp Schlatter https://orcid.org/0000-0001-9627-5903
Niclas Jansson https://orcid.org/0000-0002-5020-1631

Supplemental material

The Neko framework and the details for the test cases can be found on github. The Neko package can be downloaded here https://github.com/ExtremeFLOW/neko and the test cases on this link https://github.com/ExtremeFLOW/MSA-tests.

References

Abdelfattah A, Barra V, Beams N, et al. (2021) GPU algorithms for efficient exascale discretizations. *Parallel Computing* 108: 102841.

AlOnazi A, Keyes D, Lastovetsky A, et al. (2015) Design and optimization of openfoam-based CFD applications for hybrid and heterogeneous HPC platforms. *Arxiv*. doi: 10.48550/arXiv.1505.07630.

Arima E, Comprés AI and Schulz M (2022) On the convergence of malleability and the HPC powerstack: exploiting dynamism in over-provisioned and power-constrained HPC systems *International Conference on High Performance Computing*. NY: Springer, 206–217.

Atchley S, Zimmer C, Lange J, et al. (2023) Frontier: exploring exascale *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–16.

Borrell R, Dosimont D, Garcia-Gasulla M, et al. (2020) Heterogeneous CPU/GPU co-execution of CFD simulations on the POWER9 architecture: application to airplane aerodynamics. *Future Generation Computer Systems* 107: 31–48.

Calore E, Gabbana A, Schifano SF, et al. (2019) Optimization of lattice Boltzmann simulations on heterogeneous computers. *The International Journal of High Performance Computing Applications* 33(1): 124–139.

Deville MO, Fischer PF, Fischer PF, et al. (2002) *High-order methods for incompressible fluid flow*. Cambridge University Press, Vol. 9.

El Khoury GK, Schlatter P, Noorani A, et al. (2013) Direct numerical simulation of turbulent pipe flow at moderately high Reynolds numbers. *Flow, Turbulence and Combustion* 91(3): 475–495.

Fischer PF (2015) Scaling limits for PDE-based simulation *22nd AIAA Computational Fluid Dynamics Conference*, 3049.

Fischer PF, Lottes JW and Kerkemeier SG (2008) nek5000 Web page. Available at: https://nek5000.mcs.anl.gov.

Fischer P, Min M, Rathnayake T, et al. (2020) Scalability of high-performance PDE solvers. *The International Journal of High Performance Computing Applications* 34(5): 562–586.

Ivanov I, Gong J, Akhmetova D, et al. (2015) Evaluation of parallel communication models in nekbone, a nek5000 mini-application. In: *2015 IEEE International Conference on Cluster Computing*. IEEE, pp. 760–767.

Iyer KP, Scheel JD, Schumacher J, et al. (2020) Classical 1/3 scaling of convection holds up to $Ra = 10^{15}$. *Proceedings of the National Academy of Sciences* 117(14): 7594–7598.

Jansson N, Karp M, Perez A, et al. (2023) Exploring the ultimate regime of turbulent Rayleigh–Bénard convection through unprecedented spectral-element simulations *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–9.

Jansson N, Karp M, Podobas A, et al. (2024) Neko: a modern, portable, and scalable framework for high-fidelity computational fluid dynamics. *Computers & Fluids* 275: 106243.

Ju Y, Li M, Perez A, et al. (2023) In-situ techniques on GPU-accelerated data-intensive applications *2023 IEEE 19th International Conference on E-Science (E-Science)*. IEEE, 1–10.

Karniadakis GE, Israeli M and Orszag SA (1991) High-order splitting methods for the incompressible Navier-Stokes equations. *Journal of Computational Physics* 97(2): 414–443.

Karp M, Jansson N, Podobas A, et al. (2022) Reducing communication in the conjugate gradient method: a case study on high-order finite elements *Proceedings of the Platform for Advanced Scientific Computing Conference*, 1–11.

Karp M, Massaro D, Jansson N, et al. (2023) Large-scale direct numerical simulations of turbulence using GPUs and modern Fortran. *The International Journal of High Performance Computing Applications* 37(5): 487–502.

Karypis G, Schloegel K and Kumar V (2003) Parmetis In: *Parallel Graph Partitioning and Sparse Matrix Ordering Library. Version 2*.

Kim J, Dally WJ, Scott S, et al. (2008) Technology-driven, highly-scalable dragonfly topology. *ACM SIGARCH - Computer Architecture News* 36(3): 77–88.

Kolev T, Fischer P, Min M, et al. (2021) Efficient exascale discretizations: high-order finite element methods. *The International Journal of High Performance Computing Applications* 35(6): 527–552.

Kooij GL, Botchev MA, Frederix EM, et al. (2018) Comparison of computational codes for direct numerical simulations of turbulent Rayleigh–Bénard convection. *Computers & Fluids* 166: 1–8.

Krause D (2019) JUWELS: modular tier-0/1 supercomputer at the Jülich supercomputing centre. *Journal of Large-Scale Research Facilities JLSRF* 5: A135.

Krause D and Thörnig P (2018) JURECA: modular supercomputer at Jülich supercomputing centre. *Journal of Large-Scale Research Facilities JLSRF* 4: A132.

Kreuzer A, Eicker N, Amaya J, et al. (2018) *Application Performance on a Cluster-Booster System*. IEEE, 69–78. URL. DOI: 10.1109/IPDPSW.2018.00019.

Kreuzer A, Lippert T, Suarez E, et al. (2021) *Porting Applications to a Modular Supercomputer-Experiences from the Deepest Project*. Technical report, Jülich Supercomputing Center.

Liu X, Zhong Z and Xu K (2016) A hybrid solution method for CFD applications on GPU-accelerated hybrid HPC platforms. *Future Generation Computer Systems* 56: 759–765.

Markov S, Petkov P and Pavlov V (2019) Large-scale molecular dynamics simulations on modular supercomputer architecture with gromacs. In: *International Conference on Variability of the Sun and Sun-like Stars: From Asteroseismology to Space Weather*. Springer, 359–367.

Merzari E, Hamilton S, Evans T, et al. (2023) Exascale multiphysics nuclear reactor simulations for advanced designs. In: *Proceedings of the International Conference for High*

Performance Computing, Networking, Storage and Analysis, 1–11.

Niemeyer KE and Sung CJ (2014) Recent progress and challenges in exploiting graphics processors in computational fluid dynamics. *The Journal of Supercomputing* 67: 528–564.

Offermans N, Marin O, Schanen M, et al. (2016) On the strong scaling of the spectral element solver nek5000 on petascale systems. In: *Proceedings of the Exascale Applications and Software Conference 2016*, 1–10.

Riedel M, Sedona R, Barakat C, et al. (2021) Practice and experience in using parallel and scalable machine learning with heterogenous modular supercomputing architectures. In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 76–85.

Shpiner A, Haramaty Z, Eliad S, et al. (2017) Dragonfly+: low cost topology for scaling datacenters. In: *2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*. IEEE, 1–8.

Slotnick JP, Khodadoust A, Alonso J, et al. (2014) CFD vision 2030 study: a path to revolutionary computational aerosciences. Technical report.

Suarez E, Eicker N and Lippert T (2019) Modular supercomputing architecture: from idea to production. In: *Contemporary High Performance Computing*. CRC Press, 223–255.

Tufo HM and Fischer PF (1999) Terascale spectral element algorithms and implementations. In: *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 68–81.

van Rees WM, Leonard A, Pullin D, et al. (2011) A comparison of vortex and pseudo-spectral methods for the simulation of periodic vortical flows at high Reynolds numbers. *Journal of Computational Physics* 230(8): 2794–2805.

Witherden FD, Farrington AM and Vincent PE (2014) Pyfr: an open source framework for solving advection–diffusion type problems on streaming architectures using the flux reconstruction approach. *Computer Physics Communications* 185(11): 3028–3040.

Zhong Z, Rychkov V and Lastovetsky A (2014) Data partitioning on multicore and multi-GPU platforms using functional performance models. *IEEE Transactions on Computers* 64(9): 2506–2518.

Author biographies

Martin Karp is a Postdoctoral Researcher at the department of Engineering Mechanics at KTH Royal Institute of Technology. His research interests are primarily related to computational science and the interplay between algorithms and computer hardware. The application area of his research are large-scale simulations of turbulence. He is one of the primary developers of Neko, a scalable and portable solver for high-fidelity fluid dynamics. He holds a PhD in Computer Science from KTH Royal Institute of Technology.

Estela Suarez is Joint Lead of the Department Novel System Architecture Design at the Jülich Supercomputing Centre, which she joined in 2010. Since 2022 she is also Associate Professor of High Performance Computing at the University of Bonn, and member of the RIAG (Research and Innovation Advisory Board from EuroHPC JU). Her research focuses on HPC system architecture and codesign. As leader of the DEEP project series she has driven the development of the Modular Supercomputing Architecture, including hardware, software and application implementation and validation. She also leads the codesign efforts within the European Processor Initiative. She holds a PhD in Physics from the University of Geneva (Switzerland) and a Master degree in Astrophysics from the University Complutense of Madrid (Spain).

Jan H. Meinke received his Ph.D. in physics from Michigan State University in 2002. In his thesis work he studied the ground state behavior of disordered systems using graph algorithms. In 2005, he started to explore biological problems using Monte Carlo and other methods as a postdoc in the NIC research group Computational Biology and Biophysics. Since 2008, he has been a staff scientist of the Simulation and Data Laboratory Biology at the Jülich Supercomputing Centre. His research interests include protein folding and how to make efficient use of HPC hardware for solving scientific problems.

Måns Andersson is a Doctoral Student in Computer Science with specialization in High-Performance Computing at KTH Royal Institute of Technology with a background in numerical analysis.

Philipp Schlatter (from Zürich, Switzerland) obtained a degree in Mechanical Engineering from the Swiss Federal Institute of Technology (ETH Zürich) in 2001, and a PhD in Fluid Mechanics at the Institute of Fluid Dynamics (IFD) from ETH in 2005. He then moved to the Royal Institute of Technology (KTH) in Stockholm, first as a Postdoc, from 2007 to 2010 as an assistant professor, from 2010 to 2018 as associate professor, and from 2019 as full professor at KTH, with special interest in large-scale simulations of turbulent flows, mainly in wall-bounded configurations. In 2014 he was chosen as a Wallenberg Academy Fellow (which was extended in 2018), a prestigious programme with 5 + 5 years funding. He was also the director of the Linné FLOW Centre at KTH Stockholm, leading the fluid-dynamics community in the Swedish e-Science Research Centre, and the Swedish National Allocation Committee. In 2023 he moved to the Institute of Fluid Mechanics (LSTM) at the Friedrich-Alexander-Universität (FAU) Erlangen-Nürnberg. He is also adjunct professor at the University of

Bologna. The current research involves both large-scale simulations based on highly accurate spectral and spectral-element methods, but also close interaction to experimentalists in an effort to cross-validate simulation and experimental data.

Stefano Markidis is a Professor in Computer Science with specialization in high-performance computing systems, including supercomputers and quantum computers. Markidis holds a Ph.D. from the University of Illinois at Urbana-Champaign and an MS from Politecnico di Torino. Before joining KTH, Markidis was a researcher at the Los Alamos National Laboratory, Lawrence Berkeley National Laboratory, and KULeuven. Markidis was awarded two R&D Awards in 2005 and 2017.

Niclas Jansson is a Researcher at PDC Center for High-Performance Computing, KTH Royal Institute of Technology. He received his MSc in computer science and PhD in numerical analysis from KTH Royal Institute of Technology. His research interests include scalable numerical methods and adaptive finite and spectral element methods. He has extensive experience in extreme-scale computing as a developer of RIKEN's multiphysics framework CUBE, the HPC branch of FEniCS, and the next-generation spectral element framework Neko.