epcc

Porting and Performance Tuning of SeisSol on Multiple HPC Architectures

Zakariya Oulhadj

August 15, 2025

MSc in High Performance Computing
The University of Edinburgh
2025

Abstract

The rising complexity of modern high-performance computing systems presents a significant challenge for scientific computing. Differences in hardware architectures and software environments require extensive, system-specific tuning to achieve high performance. As a result, these obstacles can make it difficult for domain experts to run large-scale simulations efficiently, limiting the pace and scope of scientific research and discovery. In this work, we explore these challenges through a case study of SeisSol, an earthquake simulation code, by porting and tuning its performance across three different HPC systems. On the Bridges-2 system, as part of the Student Cluster Competition, we improved simulation performance by tuning the hybrid parallelism configuration and replacing suboptimal communication libraries. On ARCHER2, we further examined optimal hybrid parallelism strategies and identified a major bottleneck during the initialisation phase at scale. This was resolved by replacing the default communication backend, resulting in a substantial reduction in time-to-solution. Finally, on the GPU-based TeamEPCC cluster, we ported and addressed performance variability by updating a key communication library. These findings reinforce that performance is not automatically portable, and that architecture-aware tuning is essential for achieving scalable and efficient simulations across heterogeneous HPC environments.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor, Xu Guo, for her invaluable guidance and support throughout this project.

I would also like to extend my thanks to my fellow teammates from TeamEPCC for their collaboration during the ISC Student Cluster Competition 2025. Their dedication and teamwork made the experience both rewarding and enjoyable.

I am deeply grateful to my family and friends for their unwavering support throughout my studies. Without them, I would not be where I am today.

I dedicate this to my late grandad,

Říša Minařík

Contents

List of Tables

List of Figures

Chapter 1

Introduction

High-Performance Computing (HPC) has, since its inception over half a century ago, played a pivotal role in addressing some of the most complex scientific challenges [1]. By harnessing the collective power of hundreds–or even thousands–of interconnected computers, supercomputers enable researchers to tackle problems that would otherwise be computationally infeasible. These capabilities underpin advances in diverse fields, including climate modelling, computational fluid dynamics (CFD), astrophysics, seismology, and chemistry.

A central concern in scientific computing is *computational performance*–the ability of a system to solve problems quickly while efficiently utilising the underlying hardware. Higher performance reduces time-to-solution, supports increasingly complex simulations, and enables more frequent simulation runs. As scientific problems grow in scale and complexity, sustaining and improving performance has become critical for both research productivity and cost-effectiveness.

The growing complexity of modern HPC systems driven by heterogeneous architectures, sophisticated memory layouts, and steadily increasing core counts–has made achieving and sustaining high computational performance increasingly challenging [2]. As physical limits have slowed the rate of raw hardware performance improvements, the focus has shifted toward software optimisation. Despite peak capabilities reaching hundreds of petaflops, many HPC applications sustain only 15–20% of theoretical peak performance [3]. Maximising efficiency therefore requires careful optimisation of algorithms, data structures, and parallelisation strategies to fully exploit available hardware resources. Moreover, the diversity of architectures, from multi-core CPUs to many-core GPUs, demands performance portability to ensure scientific applications remain efficient across platforms.

In this context, *benchmarking and performance tuning* are essential for achieving opti-

mal results. These practices allow developers to systematically identify computational and communication bottlenecks, adapt configurations to specific hardware characteristics, and maximise resource utilisation. As modern HPC platforms vary widely in design, tuning is vital not only for performance but also for scalability and portability.

This dissertation investigates the porting and tuning of the large-scale scientific application *SeisSol* across three distinct HPC platforms. The objective is to maximise performance and efficiency while adapting to the unique characteristics of each system. Despite architectural similarities between some platforms, the results reveal substantial performance variation driven by factors such as memory locality and communication strategies. The aim of this work is to evaluate performance across systems with the goal of informing tuning strategies for SeisSol on previously unexplored HPC platforms.

The remainder of this dissertation is structured as follows: Chapter 2 introduces the field of HPC and the role of benchmarking and performance tuning within scientific computing. Chapter 3 describes the porting and benchmarking methodology across three platforms. Chapter 4 details the author's experience during the ISC Student Cluster Competition 2025, focusing on initial benchmarking and tuning efforts on the Bridges-2 system. Chapter 5 extends this work to ARCHER2, presenting scaling studies and communication tuning. Chapter 6 examines GPU-based evaluation on the TeamEPCC cluster equipped with NVIDIA H100 GPUs. Finally, Chapter 7 summarises the main findings and outlines directions for future work.

Chapter 2

Background

High Performance Computing (HPC) refers to the use of large-scale, parallel supercomputers to solve complex scientific and engineering problems across a wide range of domains. The computational capabilities of modern HPC systems have grown significantly with leading supercomputers, such as Frontier capable of exceeding one exaflop (10^{18} floating-point operations per second), marking the beginning of the exascale era [4]. This performance has enabled researchers using these systems to conduct increasingly detailed and computationally-intensive simulations, thereby accelerating scientific discovery and innovation.

The TOP500 list [5], which ranks the world's most powerful supercomputers based on benchmark performance, shows that in the past, each newly introduced top system often doubled the performance of its predecessor [6]. While studies have quantified rapid performance improvements over the last 30 years, sustaining such gains has become increasingly difficult.

Historically, performance improvements in computing were largely driven by advances in hardware–most notably by increasing the number of transistors in a processor. For many years, this steady growth in single-core performance allowed software to benefit from newer hardware generations without requiring significant changes. Moore's Law, proposed in 1965, famously predicted that transistor density would double roughly every two years, leading to exponential increases in computational power [7]. Similarly, Dennard scaling states that as transistors became smaller, their power density would remain constant, enabling clock frequencies to increase without a proportional rise in power consumption. Together, these trends fuelled decades of rapid performance growth. However, as transistor dimensions now approach physical and thermal limits, both Moore's Law and Dennard scaling have slowed causing traditional single-core performance scaling to plateau [8, 9].

To maintain progress despite the breakdown of frequency scaling, performance improvements now predominantly arise from exploiting parallelism at multiple levels– from instruction-level parallelism (ILP), where independent instructions are executed simultaneously within a single core, to system-level parallelism, where thousands of compute nodes collectivey solve a specific problem. This hierarchy spans SIMD vector units within a core, multiple threads and cores within a processor, multiple processors within a node, and distributed-memory parallelism across nodes.

Processor vendors have now encorprate architectures that maximise this parallelism, integrating large numbers of processing units alongside increasingly complex memory hierarchies and high-speed interconnects. This shift has fundamentally changed the optimisation challenge: rather than benefiting automatically from faster single cores, applications must now be explicitly parallelised and tuned to exploit the available concurrency. Achieving high performance requires a efficiently utilisng this complex system topology, communication patterns, memory access behaviour, and the interactions between software and hardware–factors that vary widely between platforms and complicate performance portability [10].

2.1 Modern HPC Architectures

Today, large-scale HPC systems typically employ a hybrid memory architecture, combining *shared memory* within each compute node and *distributed memory* across nodes.

2.1.1 Shared-Memory

Individual compute nodes are often described as shared-memory systems, as all processor cores within a node can access a common physical memory address space. A typical shared-memory node comprises four key components: CPUs – one or more central processing units, each typically containing hundreds of cores capable of executing independent threads; Memory – a memory pool accessible by all processors within the node, often organised into multiple memory channels to increase bandwidth; Interconnect – a high-speed internal bus or interconnect fabric (e.g., AMD's Infinity Fabric) that links processors to each other and to the memory subsystem; and the I/O Subsystem – interfaces that connect to storage devices, networking hardware, and accelerators such as GPUs.

A core architectural feature of modern HPC nodes is the Non-Uniform Memory Access (NUMA) design. NUMA arose as a response to the scalability limits of traditional shared-memory systems, where all processors accessed main memory through a sin-

gle, shared bus. As core counts per node grew, this bus became a major performance bottleneck due to contention and limited bandwidth.

In a NUMA architecture, each CPU socket is equipped with its own local memory controller and associated memory channels, allowing cores within that socket to access nearby ("local") memory at lower latency and higher bandwidth. More recent processor designs further subdivide a single socket into multiple NUMA regions, each with its own dedicated memory resources. While this increases available bandwidth and reduces contention, it also introduces variability in access times depending on a thread's location relative to the memory it accesses. Non-local memory accesses can incur significantly higher latency and lower bandwidth, making careful thread and data allocation essential for performance.

From a software perspective, NUMA-aware scheduling has therefore become an increasingly important area of research–particularly as multi-tiered NUMA topologies become more common in high-core-count HPC nodes [11].

2.1.2 Distributed-Memory

In a distributed-memory model, each compute node has its own private memory, inaccessible to processors on other nodes. Communication takes place explicitly over a high-speed interconnect such as InfiniBand, HPE Slingshot, or Intel Omni-Path, typically using message-passing protocols. This architecture scales efficiently to very large systems by avoiding the bandwidth and contention limits of global shared memory. Each node works independently on a portion of the overall problem, exchanging data with others only when necessary. However, the responsibility for managing these data transfers lies with the programmer, making performance highly dependent on interconnect characteristics–latency, bandwidth, and topology–as well as the efficiency of the application's communication patterns.

2.1.3 Graphics Processing Units (GPUs)

In addition to CPUs, modern HPC nodes increasingly incorporate specialised accelerators, most notably Graphics Processing Units (GPUs), as part of a heterogeneous architecture. This trend is driven by the need to achieve exascale performance while maintaining energy efficiency. Unlike CPUs, which are optimised for low-latency execution of a small number of complex threads, GPUs are designed for high-throughput execution of thousands of lightweight threads in parallel.

The adoption of GPU-accelerated systems in HPC has grown steadily over the past decade. In the TOP500 rankings, GPU-based hardware accounted for only 28% of sys-

tems between 2011 and 2019; by November 2024, this had risen to 210 systems (42% of the list), representing a further 3.2% increase in just six months [6]. GPUs offer extremely high memory bandwidth and computational throughput, but fully exploiting their potential requires accelerator-specific programming models (e.g., CUDA, HIP, OpenACC), efficient data movement between host and device memory, and careful performance tuning to the underlying architecture.

2.2 Parallel Programming Models

To fully utilise the distributed-memory architectures of modern HPC systems, two parallel programming paradigms are most commonly employed: Open Multi-Processing (OpenMP) [12] for shared-memory parallelism within a node, and the Message Passing Interface (MPI) [13] for communication across nodes.

OpenMP provides a standard Application Programming Interface (API) for shared-memory parallelism, implemented through compiler preprocessor directives. It enables multiple threads (running on cores) to operate concurrently within the same address space, supporting fine-grained parallelism over loops and code regions. This approach can deliver substantial performance gains, particularly for computationally intensive workloads, and includes features such as tasking, thread affinity, and memory binding–important for optimising performance on NUMA-based systems.

In contrast, MPI is a widely adopted standard for distributed-memory parallelism [14]. It enables explicit communication between processes through message passing and is available in several popular implementations, including OpenMPI, MPICH, MVAPICH, and Cray MPI. MPI forms the backbone of inter-node communication in most HPC applications and is critical for scaling to large numbers of nodes on modern supercomputers.

Many applications are increasingly employing a hybrid programming model–commonly referred to as MPI+OpenMP [15]–in which both paradigms are used simultaneously: MPI manages inter-node communication, while OpenMP exploits shared-memory parallelism within each node. Performance studies have shown that hybrid codes can significantly outperform pure MPI implementations at higher core counts by reducing communication overhead and memory footprint [16, 17].

2.3 Benchmarking and Profiling Tools

Effectively applying parallel programming models can be challenging–not only in achieving correct implementation, but also in attaining high performance. Systematic performance evaluation through benchmarking and profiling is therefore essential for understanding application behaviour on HPC systems, identifying bottlenecks, and guiding targeted optimisation.

Benchmarking evaluates hardware and software performance to assess scalability, identify performance limitations, and determine the most efficient resource configurations. Key metrics include *strong scaling*, which measures performance gains as resources increase for a fixed problem size, and *parallel efficiency*, which quantifies how close performance comes to ideal linear speedup.

While benchmarking reveals an application's performance characteristics, profiling explains *why*. Profilers analyse resource utilisation, computational hotspots, MPI communication overheads, and memory access patterns, enabling targeted optimisation. For CPU-based performance tuning, widely used tools include Linaro MAP [18], which provides detailed function-level runtime and memory access analysis, and CrayPat [19], part of the Cray Performance Measurement and Analysis Tools suite, which offers in-depth performance statistics, MPI communication analysis, and hardware counter data to identify bottlenecks. For GPU-accelerated systems, vendor-provided tools such as NVIDIA Nsight Systems and Nsight Compute deliver detailed system and kernel-level analysis [20, 21].

The concepts of parallel programming, benchmarking, and profiling form the foundation for understanding and tuning real-world HPC applications which are explored through SeisSol.

2.4 SeisSol: An Earthquake Simulator

SeisSol (Seismic Solver) is a state-of-the-art simulation tool designed for the accurate and high performance modelling of earthquake processes and seismic wave propagation [22]. It has been developed through ongoing collaboration between researchers at the Technical University of Munich (TUM) and Ludwig Maximilian University of Munich (LMU). Originating from research into high-order methods for seismic wave propagation, it was recognised as a Gordon Bell Prize finalist in 2014, highlighting its excellence in scalable scientific computing and its contribution to advancing performance in real-world seismic simulations [23]. Since 2018, SeisSol has been part of the ChEESE Centre of Excellence project (Phase 1), used to demonstrate exascale readiness for seismic modelling, particularly for earthquake and tsunami simulations [24].

SeisSol has been applied across various domains, including seismic hazard assessment and oil and gas exploration. It enabled the first 3D dynamic rupture simulation of the Hellenic Arc megathrust earthquake [25] and was used to study the 2019 Ridgecrest event [26], demonstrating that shallow damaged fault zones can enhance ground motion and enable supershear rupture. Recent works using the application have also revealed that complex, cascading fault networks can significantly amplify high-frequency shaking and peak ground accelerations [27], highlighting the critical role of fault complexity in seismic hazard analysis.

SeisSol numerically solves the elastic wave equations using a high-order Discontinuous Galerkin (DG) method for spatial discretisation on unstructured tetrahedral meshes, combined with the Arbitrary high-order DERivatives (ADER) scheme for time integration [28]. To further enhance efficiency, it employs a multi-rate local time-stepping (LTS) method, where elements are grouped into clusters based on their local time step [29]. This allows more dynamic regions to be updated more frequently, reducing overall runtime by up to 2.6x compared to traditional global time stepping. The ADER-DG formulation is particularly well-suited for capturing complex seismic wave propagation and dynamic fault rupture on large-scale, parallel architectures.

A key component of SeisSol is its use of of external code generators during compilation. The YATeTo library maps the ADER-DG scheme, written in a domain-specific language (DSL), to Loop-over-GEMM (General Matrix Multiply) operations [30]. These are then compiled into highly optimised C/C++ or inline assembly using specialised backends: LIBXSMM for small dense matrix-matrix multiplications, PSpaMM for sparse matrices, and Eigen as a general-purpose fallback [31, 32, 33]. These backends produce hardware-specific instructions targeting SIMD vector extensions such as SSE, AVX, AVX2 as well as AVX-512, enabling high throughput on modern CPUs and is attributed to the high performances achieved during simulations.

Overall, SeisSol has been extensively optimised for modern HPC architectures, supporting both CPU and GPU execution [34, 35], and employs an MPI+X hybrid parallelisation model–using OpenMP on CPUs and CUDA on GPUs. Previous studies have demonstrated that this approach enables SeisSol to achieve petascale performance, reaching 1.09 PFLOP/s on systems such as SuperMUC [36]. MPI communication is performed asynchronously to overlap computation with data exchange, with two configurable progression strategies: threaded and polling. By default, SeisSol adopts the threaded approach, in which each MPI rank spawns a dedicated communication thread to manage asynchronous message progression in the background [37]. In contrast, the polling mode omits the dedicated thread, relying instead on the application periodically checking for message progress during the simulation.

2.4.1 Simulation Benchmarks

A variety of benchmark scenarios are available to verify correctness, validate output, and to benchmark SeisSols's performance. These include a lightweight kernel proxy, precomputed reference simulations, and a large-scale real-world earthquake event.

Lightweight Proxy Tool

The proxy application is a lightweight tool designed to generate artificial test scenarios for two main purposes. First, it verifies the correctness of all code-generated numerical kernels by ensuring they execute without errors. Second, it isolates and evaluates the idealised peak performance of the core ADER-DG simulation kernels on a single node, excluding external factors such as MPI communication and I/O overhead. Using this tool requires defining the number of cells (problem size), how many timesteps the simulation should run for and lastly, which micro-kernel should be benchmarked.

Precomputed Reference Scenarios: The Problem Version 33

A set of precomputed earthquake scenarios is available to validate SeisSol by comparing simulation output against established reference solutions [38]. One widely used test case is *The Problem Version 33 (TPV33)*, a benchmark developed by the Southern California Earthquake Center (SCEC) [39]. It simulates a spontaneous rupture on a vertical strike-slip fault embedded in a low-velocity fault zone within a linear elastic medium. TPV33 is commonly used to evaluate the physical accuracy and numerical stability of dynamic rupture models across different simulation codes [40].

Türkiye Kahramanmaraş Scenario

Our performance investigations will be largely utilising the Türkiye Benchmark scenario, based on a real-world event officially referred to as the "Türkiye Kahramanmaraş earthquake doublet". This catastrophic sequence involved two major earthquakes that struck southeastern Türkiye (formerly known internationally as Turkey) on the 6th February 2023, the first occurring at 01:17 UTC near the Türkiye–Syria border. The two events had moment magnitudes of M_{w} 7.8 and M_{w} 7.7, occurring approximately nine hours apart along the East Anatolian Fault (EAF) and Sürgü–Misis Fault, respectively.

The initial rupture, which is the primary focus of the simulation, originated on the Narli fault, propagating northeast and reaching the EAF junction after approximately

11.6 seconds. It then expanded bilaterally along the EAF in an East–West direction: the southwest rupture, along the Amanos segment, persisted for 78 seconds, while the northwest rupture, along the Yabaşi and Erkenek segments, lasted 60 seconds. In total, this event ruptured over 350 km of fault length [41].

To gain deeper insight into the geophysical processes during the event, dynamic rupture and seismic wave propagation are accurately modelled using SeisSol. The benchmark dataset was originally published by Jia et al. and Gabriel et al. (2023) [42] and subsequently adapted into a SeisSol-compatible format by Thomas Ulrich [43]. The simulation is configured via the `parameters.par` file, which sets key options such as the simulation duration (defaulted to 10 seconds), the use of a Linear Slip-Weakening friction law (FL = 16) to characterise fault behaviour, and a discretisation scheme featuring Local Time Stepping (ClusteredLTS = 2) for enhanced computational efficiency. The file also specifies input mesh paths and output controls to manage data generation.

In addition to the main parameter file, the setup includes eight YAML files that define the scenario's initial conditions–such as material parameters, initial fault stress distribution, and prescribed rupture time. These configuration files, along with the mesh, are located within the mesh subdirectory. The primary mesh file, `Turkey78_75_dip70_2`, is approximately 103 MB in size and contains 2,410,756 elements.

The mesh is defined in a Cartesian coordinate system with units in metres, spanning approximately 204,491m, 217,824m, and 21,873m along the X, Y, and Z axes, respectively. The directory also includes a `Mw_78_Turkey_faultreceivers.dat` file, which defines the locations of fault receivers. These receivers are used to record synthetic seismograms, capturing rupture time, fault slip, and resulting ground motion at specified locations throughout the simulation domain.

Chapter 3

Methodology

The methodology defines a framework for porting, benchmarking, and tuning SeisSol across different HPC systems. It outlines the general process–from software compilation and kernel verification to benchmarking and performance tuning–while maintaining consistent simulation parameters. Given substantial hardware and software differences, the system-specific procedures and optimisations are detailed in their respective chapters.

3.1 Overview of HPC Systems Used

Our performance investigation spans three HPC systems. The first, Bridges-2, was used in the Student Cluster Competition, where the primary objective was to maximise performance on four CPU nodes. The analysis was then extended to ARCHER2 to examine SeisSol's scalability and efficiency across a larger number of nodes. The third system, the TeamEPCC cluster, is a GPU-accelerated platform equipped with NVIDIA H100 GPUs, where the focus was on evaluating SeisSol's performance on modern GPU hardware in light of its recent GPU support.

3.2 Process of Porting SeisSol

Software Compilation Porting SeisSol onto each system first consisted of identifying the key programming environments that could be utilised. This included the selection of compiler toolchains and MPI communication libraries. For each programming environment identified, SeisSol and all of its dependencies would be

compiled manually using the same compiler toolchain for consistency. Dependencies made use of a mixture of Makefiles and CMake build systems with each dependency being compiled as part of an individual bash script. For each compiled dependency, a corresponding Lmod modulefile would be created based on the dependency name and version to allow individual dependencies to be easily loaded/unloaded when compiling SeisSol. SeisSol itself was compiled using the latest version, v1.3.1, and the CMake build system and followed the same compilation methodology as the dependencies. Simulation-specific options for all systems were kept consistent based on the initial competition rules, which used double precision and a convergence order of four.

Kernel Generator Selection Following successful compilation, SeisSol's single-node proxy application was used for two primary purposes: (1) to verify the correctness of the build by ensuring that all numerical kernels executed without error, and (2) to benchmark different compilers and kernel generator combinations to identify the optimal single-node configuration. This served as the baseline for selecting the configuration used in subsequent benchmarking and tuning.

On Bridges-2, strong scaling was performed by keeping the problem size constant but increasing the number of cores used until a single full node was utilised. After the competition, to align with previous investigations, on ARCHER2, strong scaling benchmarks were conducted by varying the number of elements from 512 to 524,288, doubling at each step. For each problem size, a full node was utilised with one thread per physical core. On the TeamEPCC cluster, a single GPU was used.

Each compiler-kernel configuration was executed three times, and the results were averaged to reduce run-to-run variability.

TPV33 Verification Check Verification was extended to the TPV33 benchmark, which includes reference solutions provided by the SeisSol developers. This ensured that all components of SeisSol functioned correctly, including mesh initialisation, the main simulation, and output generation.

The test was executed on a full single node using a convergence order of six, matching the configuration used to generate the official reference solutions. SeisSol's source code includes a `compare-receivers.py` Python script, which was used to compare the simulation outputs against the precomputed reference data. The script computes the relative L2-norm error for each output receiver file. Two arguments were passed to the script: `--prefix tpv5` to specify the file naming prefix for TPV33, and `--epsilon 0.01` to set the error tolerance, flagging any discrepancies exceeding a threshold of 0.01.

3.3 Benchmarking Tests and Tunings

Evaluating SeisSol's performance was conducted by benchmarking and utilising both the smaller TPV33 and the larger Türkiye scenario initially provided as part of the ISC SCC25 competition, focusing on two key metrics. The flop-rate, reported as HW-GFLOP/s (Hardware GFLOPS/s) by SeisSol during and at the end of the simulation that indicates the total number of hardware floating-point operations executed during the simulation. For ARCHER2 and the TeamEPCC cluster specifically, we also measured and analysed the parallel efficiency, which indicates how efficiently the hardware resources are being utilised based on the speedup. The formula used is:

$$E(N) = \frac{S(N)}{N} = \frac{F(N)}{N \cdot F(1)} \tag{3.1}$$

where $E(N)$ is the efficiency at N nodes, $F(N)$ denotes the achieved performance at N nodes, $F(1)$ is the single-node performance, and $S(N)$ is the speedup.

NUMA Effects for Process and Thread Counts For the CPU-based systems (Bridges-2 and ARCHER2), we assessed the impact of NUMA on SeisSol's performance with its hybrid MPI+OpenMP model. We varied the number of MPI ranks per node and OpenMP threads per rank to identify the configuration yielding the highest flop rate and parallel efficiency. Experiments were performed across increasing node counts to determine whether NUMA-related effects became more pronounced at larger scales, with all cores fully utilised in every case. On Bridges-2, tests were limited to a single node using the smaller TPV33 scenario due to prolonged queue times, whereas on ARCHER2 we employed the larger Türkiye benchmark across multiple node counts.

Communication Mode and Affinity Masking SeisSol supports two communication modes: *polling* and *threaded*. Using the Türkiye benchmark, we conducted strong-scaling tests on Bridges-2, ARCHER2, and the TeamEPCC cluster to determine the most effective mode. For the threaded configuration, we also assessed the impact of explicitly setting affinity masks, comparing flop rates with and without masking to evaluate potential gains in communication efficiency.

Communication Backends On Bridges-2, multiple MPI implementations (e.g., Open-MPI, MVAPICH2) were benchmarked to determine the most performant for the competition's four-node target configuration. This included building a custom optimised OpenMPI version. On ARCHER2, as only a single MPI implementation (Cray MPICH) is available, we instead benchmarked alternative communication backends such as OFI and UCX. On TeamEPCC, we investigated two versions of the HPC-X communication library.

Cross-System Comparisons Comparative analyses were conducted to evaluate performance differences between the CPU-based systems, Bridges-2 and ARCHER2, by measuring the flop rate over ten simulated seconds of the Türkiye benchmark at one, two, and four nodes. Additionally, single-node results from ARCHER2 were compared against single-GPU runs on the TeamEPCC cluster to assess the idealised proxy results.

3.4 Profiling

Our benchmarking was complemented with profiling using a variety of profiling tools across the three different systems to guide different tuning decisions. On Bridges-2, Linaro Map and Scalasca were utilised as part of one of the three tasks for the competition to identify the most frequently called MPI functions. On ARCHER2, the CrayPat profiler, was used to investigate MPI-related communication bottlenecks during the initialisation. Lastly, on the TeamEPCC cluster, NVIDIA Nsight Systems was used to evaluate performance variability.

Chapter 4

Porting and Performance Optimisation on Bridges-2

This chapter outlines the benchmarking and optimisation efforts carried out on the Bridges-2 system as part of the ISC Student Cluster Competition 2025 (ISC SCC25). The focus was on maximising performance, both in terms of flop-rate and time-to-solution. A variety of tuning strategies and benchmarking experiments were conducted to evaluate and improve SeisSol's scalability within the competition's resource constraints.

4.1 The Student Cluster Competition

The ISC SCC25, organised by the HPC-AI Advisory Council [44], is a student-focused competition held annually as part of the ISC High Performance Conference [45]. It challenges teams to demonstrate their HPC skills by benchmarking and tuning real-world scientific applications to achieve optimal performance.

The competition is held in two formats, with locations varying each year. For this years edition (2025), the onsite event took place in Hamburg, Germany, while the virtual format was hosted on the Bridges-2 supercomputer at the Pittsburgh Supercomputing Centre (PSC) [46]. Representing the Edinburgh Parallel Computing Centre (EPCC) [47], our team–*TeamEPCC*–was selected to participate in the virtual format, which ran from 1st April to 11th May and featured 26 teams from institutions around the world.

The team consisted of three members, each assigned a specific application. The author was responsible for SeisSol and was tasked with completing three main objectives

during the competition:

1. Run SeisSol on four CPU nodes and submit the best-performing results. Additionally, benchmark both types of communication modes–polling and threaded–and showcase any performance differences.
2. Then, using an MPI profiler, identify the three most frequently used MPI function calls during the simulation.
3. Create a short video of the Türkiye simulation showing a visualisation using the ParaView application [48].

For these tasks, the Türkiye benchmark scenario, discussed previously (see Section 2.4.1), was provided as our input files.

4.2 System Specifications

Bridges-2 is hosted at the Pittsburgh Supercomputing Centre (PSC), a joint research facility operated by Carnegie Mellon University and the University of Pittsburgh [49]. The system comprises over 550 HPE Apollo 2000 Gen11 compute nodes and offers a mix of Regular Memory (RM), Large Memory (LM), Extreme Memory (EM) and GPU-accelerated nodes. Each RM node is equipped with two AMD EPYC™ 7742 processors (64 cores per socket, 2.25 GHz) and 256GB of RAM. All nodes are interconnected via a Mellanox HDR-200 InfiniBand network, arranged in a fat-tree Clos topology. Each node uses a Mellanox ConnectX-6 (MT28908) adapter, supporting HDR speeds of up to 200 Gbit/s. The system provides a 347 TB home partition (`/jet/`) and a larger 16 PB work partition (`/ocean/`).

The system runs the Red Hat Enterprise Linux Ootpa (RHEL) v8.10 operating system. There are five available programming environments, including AOCC v2.3.0, Intel v2021.10.0 (`icx`), Intel OneAPI v2023.2.0 (`llvm`), GNU GCC v13.3.1 and NVIDIA v22.9-0 (`nvhpc`). System dependencies were managed using modulefiles by Lmod v8.2.7, and job submissions for both batch scripts and interactive mode were managed by SLURM v22.05.11.

During the competition, each team was limited to a maximum of four RM compute nodes and granted access to both the `/jet/` and `/ocean/` partitions with a maximum allowed capacity of 11 TB. To ensure accessibility from all nodes, all benchmarking scripts, input files, and output data were stored in the `/jet/` partition.

4.3 Porting SeisSol to Bridges-2

The porting process is organised into three main stages. First, all required dependencies and the SeisSol application are compiled using the available programming environments. Second, the proxy application is used to verify the correctness of all generated numerical kernels–ensuring they execute without errors–and to evaluate idealised single-node performance in isolation from communication and I/O. Finally, the TPV33 benchmark is employed for correctness testing by comparing the simulation output against reference solutions to validate the accuracy of the installation.

Table 4.1: List of four configurations compiled for the SCC on Bridges-2 including one Intel-based and three GCC-based builds of SeisSol v1.3.1.

Components	Configuration 1	Configuration 2	Configuration 3	Configuration 4
Build Toolchains				
Compiler	ICC v2021.10.0	GCC v13.3.1	GCC v13.3.1	GCC v13.3.1
MPI	OpenMPI v4.1.7	OpenMPI v4.0.5	MVAPICH2 v2.3.0	OpenMPI v5.0.7 *
Dependencies				
SeisSol	1.3.1	1.3.1	1.3.1	1.3.1
HDF5	1.14.1	1.12.3	1.12.3	1.12.3
NetCDF	4.9.2	4.9.3	4.9.3	4.9.3
yaml-cpp	0.6.3	0.6.3	0.6.3	0.6.3
Lua	5.3.6	5.3.6	5.3.6	5.3.6
EASI	1.0	1.0	1.0	1.0
ParMETIS	4.0.3	4.0.3	4.0.3	4.0.3
Eigen	3.4.0	3.4.0	3.4.0	3.4.0
PSpaMM	1.0	1.0	1.0	1.0

* Custom built OpenMPI v5.0.7

Table 4.1 shows the four compiled configurations, detailing the compiler, MPI implementation, and software dependencies used. Multiple compilers and MPI variants were tested to evaluate the impact in terms of performance for SeisSol across different libraries.

4.3.1 Compiling SeisSol for the Intel Toolchain

The HPC Advisory Council published a SeisSol user guide titled "Getting Started with SeisSol for ISC SCC25", which provided step-by-step instructions for building SeisSol on the Bridges-2 system [50]. This guide was used as a starting point, as it offered pre-compiled dependencies built with the Intel ICC v2021.10.0 (20230609) toolchain and the HPC-X v2.22 toolkit, including OpenMPI v4.1.7 for parallel support. These dependencies were made available via modulefiles in the shared `ISC25/SeisSol` directory, streamlining the initial stages of the porting process.

While attempting to use the Intel compiler stack with HPC-X v2.22, we encountered persistent runtime errors when launching multiple MPI processes. Specifically, OpenMPI v4.1.7 reported failures during the `MPI_Init_thread` call, stating that `MPI_INIT` could not complete because one or more processes were unreachable. Similar errors occurred across multiple OpenMPI versions. Although the root cause could not be conclusively identified, the error messages suggested that essential components–such as the Byte Transfer Layer (BTL) or Matching Transport Layer (MTL)–were not properly loaded, possibly due to misconfiguration in the pre-installed OpenMPI environments. Consequently, we opted to use the GCC toolchain, which did not exhibit these issues.

4.3.2 Manually compiling SeisSol for the GCC Toolchain

The SeisSol application was compiled using GCC v13.3.1 and the GNU toolchain, with all required dependencies manually built from source.

Three different MPI implementations were evaluated:

- OpenMPI v4.0.5 – the default system-provided version available through the `openmpi/4.0.5-gcc10.2.0` module.
- MVAPICH2 v2.3.0 – supplied via the `mvapich2/2.3.5-gcc8.3.1` module.
- OpenMPI v5.0.7 – a custom, CPU-optimised version built from source and introduced later in the competition. This version was made available to all team members through a shared `team_epcc` module directory.

Most dependencies, including HDF5, yaml-cpp, Lua, EASI, ParMETIS, Eigen, as well as PSpaMM–were successfully compiled with no major issues. However, our initial attempts at building NetCDF v4.9.2 proved particularly challenging. Although the library was configured with the `--disable-dap` flag–as advised by the SeisSol documentation to disable OPeNDAP (Open-source Project for Network Data Access Protocol) support, which is not required–SeisSol still attempted to link against `libcurl.so` at runtime. This prevented the executable from launching due to the dynamic linker

being unable to locate the shared library. This unexpected behaviour required extensive debugging, including testing alternative configuration flags and experimenting with earlier versions of NetCDF to investigate whether this particular version (v4.9.2) was problematic.

The issue was ultimately traced to a bug and discussed in GitHub Issue #2473 [51], which revealed NetCDF v4.9.2 erroneously included optional dependencies even when explicitly disabled. Upgrading to NetCDF v4.9.3 resolved the problem, as this version correctly excluded unnecessary libraries such as `libcurl` when configured appropriately. As a result, subsequent attempts to run SeisSol did not exhibit the same errors.

We also attempted to compile the `libxsmm` kernel generator for SeisSol; however, we repeatly encountered "wrong usage exit errors" that could not be resolved despite troubleshooting efforts. Consequently, we were unable to evaluate alternative code generators and instead resorted to the PSpaMM Python library.

4.3.3 Compiling SeisSol

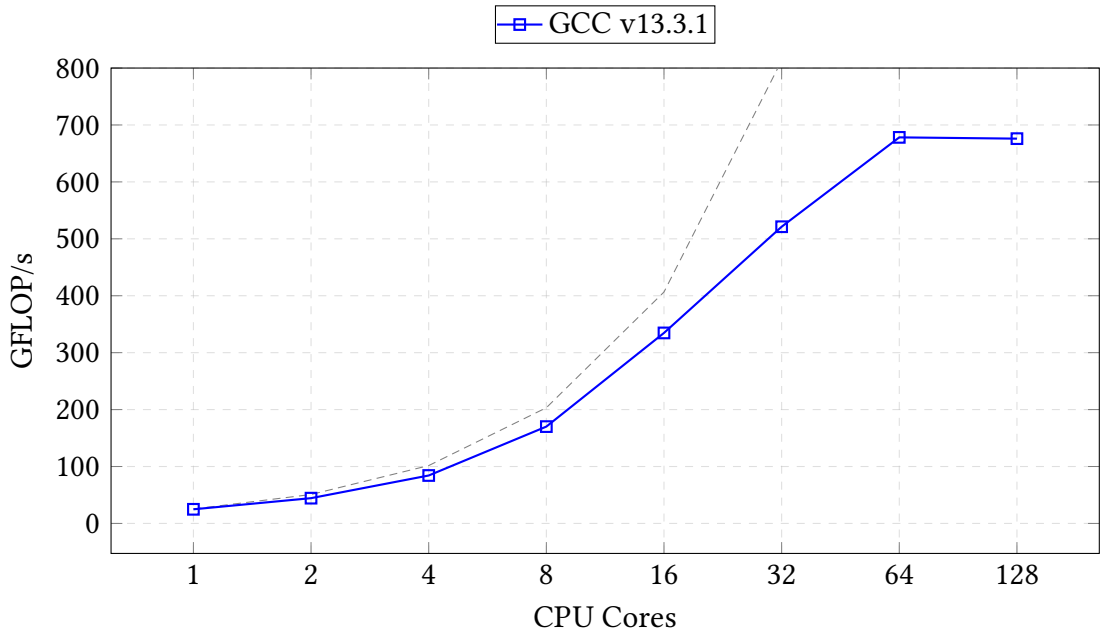
All builds were based on the latest SeisSol v1.3.1 and employed a consistent CMake configuration to ensure an equal comparison. The build type was set to `Release`, enabling the compilers to apply the highest level of optimisations `-O3`. The `-DHOST_ARCH` flag was set to `rome` which specifies the `-march=znver2` and `-mtune=znver2` flags for the Zen 2 architecture. NUMA-aware pinning was enabled to ensure correct process and thread placements on each node. The use of ASAGI was explicitly disabled, as the Türkiye benchmark input mesh was not preprocessed into an ASAGI-compatible format and therefore, not utilised. Key simulation parameters, such as numerical precision, convergence order and equation, were set to `double`, 4 and `elastic`, respectively. Graph partioning was set to `parmetis` and the kernel generator was set to `PSpaMM`. These follow both the Getting Started guide and guidance from both the competition committee and SeisSol developers.

Furthermore, we additionally attempted to build SeisSol using the AOCC v2.3.0 toolchain. However, this effort was ultimately unsuccessful due to a `std::system_error` exception encountered during the compilation of HDF5 v1.12.3. Despite multiple attempts to resolve the issue, the error could not be resolved, and thus, performance using the AOCC toolchain could not be evaluated. We suspect that this may be due to a bug in the specific AOCC version available, and so further investigation would be required using an alternative version.

4.3.4 Evaluating Idealised Single-Node Performance

For our first benchmark, we utilised the proxy using GCC v13.3.1 to evaluate whether the PSpaMM-generated kernels execute correctly and to estimate the peak single-node performance (excluding communication and I/O). We ran with one MPI rank and 128 OpenMP threads, ensuring all cores within a single node are utilised. Each thread is bound to a separate physical core by setting `OMP_PLACES=cores`. The benchmark was configured to run for 50 timesteps for a problem size of one million elements for all micro-kernels using `all`.

Figure 4.1: Idealised single-node performance of PSpaMM-generated kernels using the SeisSol proxy (excluding MPI communication and I/O), compiled with GCC v13.3.1 on Bridges-2. The benchmark was run with one million elements and 50 time steps, testing all kernel variants.



Our initial runs (Figure 4.1) confirmed that all kernels ran without errors and showed an S-shaped trend peaking at 675.97 GFLOP/s for full-node utilisation (128 cores). Beyond 16 cores, we observed performance plateauing with the difference between 64 and 128 cores being only 2.24 GFLOP/s. We suspect that this is a result of the increased memory latency due to cross-socket communication.

4.4 Task 1: Achieving the Best Performance on Four Nodes

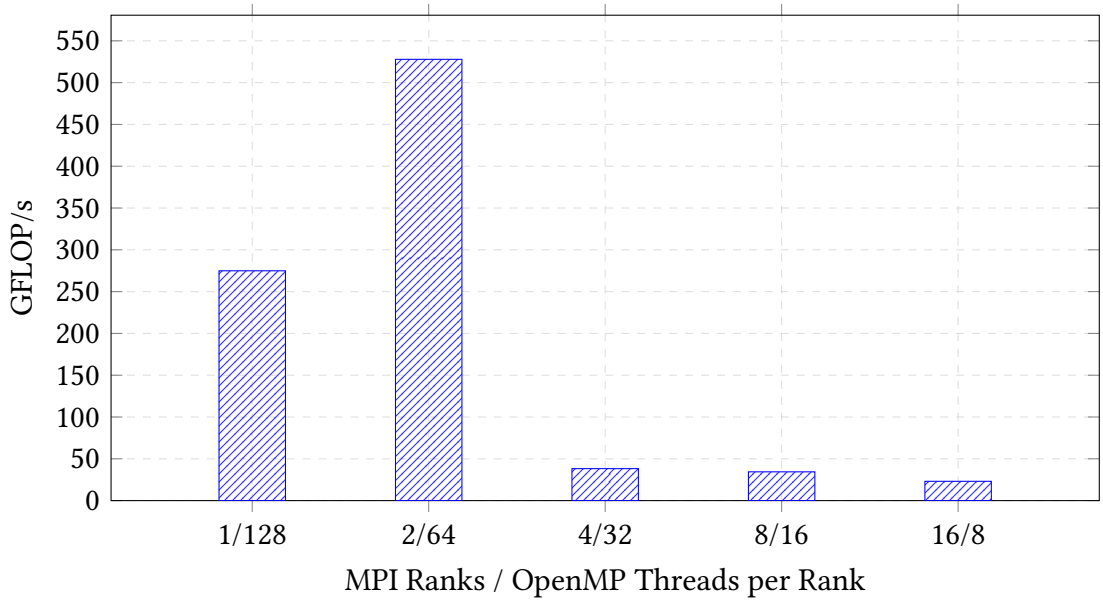
4.4.1 Verifying NUMA effects using TPV33 Benchmark

To assess potential NUMA effects, we used the `numactl -H` command to inspect the NUMA layout of Bridges-2. The output confirmed that each node comprises two NUMA domains: socket 0 is associated with cores 0–63, and socket 1 with cores 64–127.

Our NUMA performance analysis was carried out using the smaller TPV33 benchmark for two main reasons. Firstly, it allowed runs to complete within a reasonable timeframe. Secondly, attempting the larger Türkiye simulation would have required a greater number of nodes, significantly increasing SLURM queue wait times. TPV33 was therefore used to evaluate various MPI and OpenMP configurations within practical runtime limits, with the goal of identifying the best performance given the NUMA layout.

To ensure correct process and thread binding, we launched jobs with the `mpirun` command and specificed `--map-by ppr:X:node:PE=Y`, varying the number of MPI ranks (X) and OpenMP threads per rank (Y) while fully utilising one node. We also set `--bind-to core` to bind threads to physical cores. To verify that bindings were applied correctly, we enabled `--display-map` and `--report-bindings`, which displayed the sockets and cores allocated to each process.

Figure 4.2: Evaluating the single-node NUMA effects of running the TPV33 benchmark in polling communication mode by varying the number of MPI ranks and OpenMP threads (Processes/Threads) on Bridges-2.



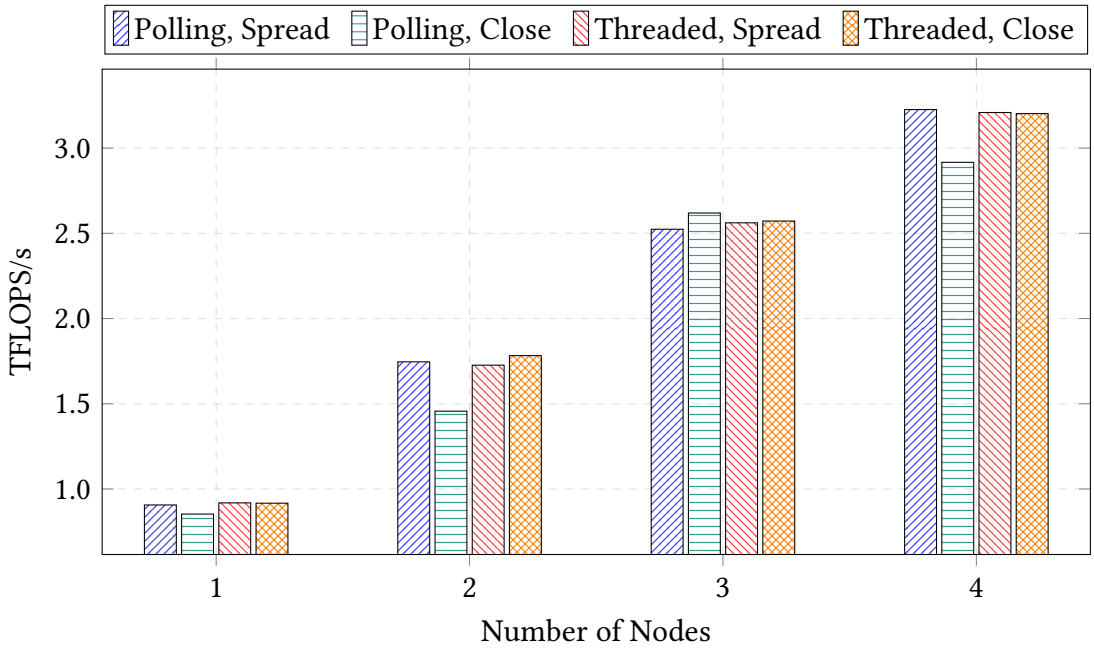
Our results, shown in Figure 4.2, indicate that the configuration with 2 MPI ranks and 64 OpenMP threads per rank achieved the best performance, reaching 527.87 GFLOP/s and a time-to-solution of 12.74 seconds. This outperformed the second-best configuration of 1 MPI rank and 128 threads per rank by 252.97 GFLOP/s and 11.68 seconds for flop-rate and time-to-solution, respectively. This indicated that placing one MPI rank per NUMA node resulted in the highest performance, aligning with the NUMA layout previously identified, ensuring that both memory allocations and accesses remain local to the socket on which a thread is running.

However, it is important to note that these results were obtained using only a single-node run of the TPV33 benchmark due to significantly long SLURM queue times. As such, it remains unclear whether the same configuration would yield optimal performance when applied to (1) the full-scale Türkiye simulation or (2) multi-node executions involving four nodes. Further investigation is required to confirm whether the observed NUMA-aware benefits generalise beyond the single-node scenario. Nevertheless, we selected the 2/64 configuration for further benchmarking.

4.4.2 Strong Scaling Benchmarking for Türkiye Scenario

We next conducted a strong scaling analysis of the Türkiye benchmark from one to four nodes comparing the performance of the *polling* and *threaded* communication modes as well as different OpenMP thread binding strategies. To enable the *threaded* communication mode, we explicitly set the `SEISSOL_COMMTHREAD=1` environment variable, and also made sure to reserve one core for each MPI rank with `OMP_NUM_THREADS=63`. This means that 63 threads are used for computing the simulation, whilst one thread is solely for progressing the MPI communication engine. For OpenMP, we ensured that threads were bound to cores using `OMP_PLACES=cores` and for thread placement strategies, we used the `OMP_PROC_BIND=close or spread` variable.

Figure 4.3: Strong scaling of the Türkiye benchmark on Bridges-2, configured with 2 MPI ranks and 64 OpenMP threads per rank across 1 to 4 nodes. The results compare performance across communication modes (polling vs. threaded) and OpenMP thread affinity strategies.



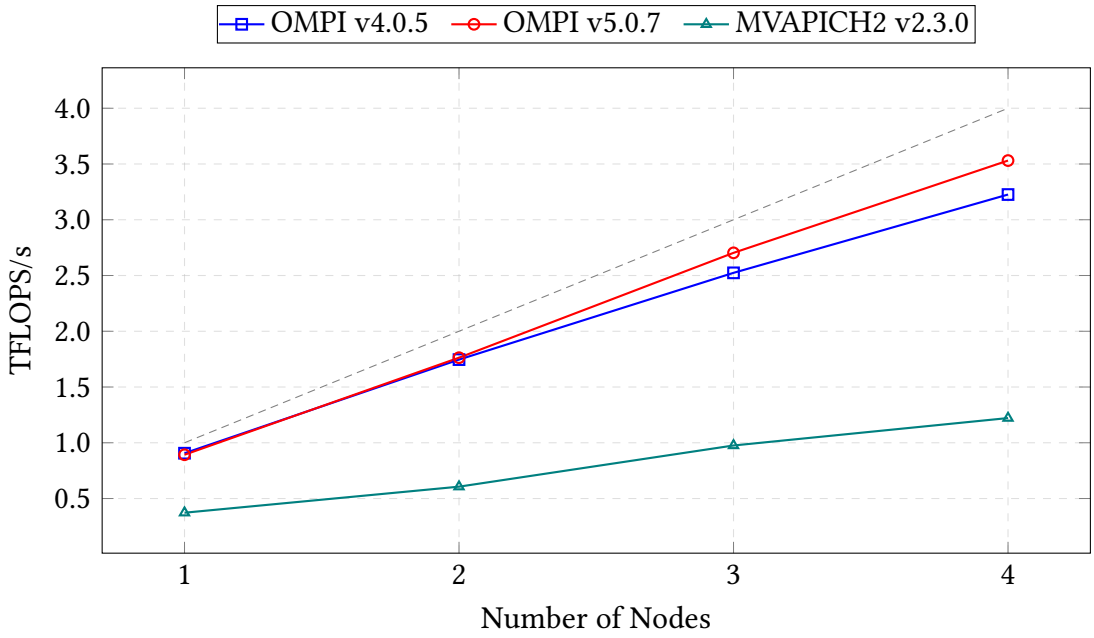
Performance for both the polling–spread and threaded–spread configurations was comparable across all node counts, whereas polling–close exhibited greater variability. Among all configurations, polling–spread achieved the highest performance, peaking at 3,226.1 GFLOP/s on four nodes–18.1 GFLOP/s higher than the next best (threaded–spread). Consequently, polling–spread was selected for subsequent experiments.

4.4.3 Comparing Alternative MPI Libraries

We further investigated other alternative MPI communication libraries, including OpenMPI v4.0.5 (the default), MVAPICH2 v2.3.0, and a custom-built OpenMPI v5.0.7. Using the `ompi_info` command, we found that the system-provided OpenMPI v4.0.5 was built via Spack with GCC v10.2.0 and relied on outdated components, including UCX v1.9.0 and HWLOC v2.2.0. However, it lacked support for the Unified Collective Communication (UCC) library, which offers optimised implementations of collective operations.

We manually compiled an optimised OpenMPI v5.0.7 with GCC v13.3.1 and linked it against more recent versions of the dependencies: UCX v1.18.0, UCC v1.3.0, and HWLOC v2.12.0. Architecture-specific optimisation flags were also applied to target the Zen 2 architecture: `--with-mcpu`, `--with-march`, `--enable-optimizations`, and `--enable-mt` to enable multithreading and maximise runtime efficiency.

Figure 4.4: Strong scaling of the Türkiye benchmark on Bridges-2 using 2 MPI ranks per node and 64 OpenMP threads per rank, scaled from 1 to 4 nodes. Results compare performance across OpenMPI v4.0.5, OpenMPI v5.0.5, and MVAPICH2 v2.3.0.



The benchmarking results of all three MPI implementations are shown in Figure 4.4, indicating that MVAPICH2 performed significantly worse, exhibiting a 2.004 TFLOP/s reduction on four nodes compared to the default OpenMPI v4.0.5. While the exact

cause remains uncertain, the degradation is likely due to a suboptimal default configuration. Given the substantial performance loss, we disregarded it and instead focused on the custom-built OpenMPI v5.0.7. In contrast, it demonstrated notable improvements beyond two nodes, delivering gains of 179.3 GFLOP/s and 304.7 GFLOP/s on three and four nodes, respectively. The maximum performance achieved was 3,530 GFLOP/s–an increase of 303.9 GFLOP/s over the previous best. This was accompanied by a reduction in total simulation time to 1,076 seconds, illustrating the benefits of using an up-to-date and optimised MPI stack tailored to the target architecture.

4.4.4 Overlapping Fault Output with Asynchronous I/O

The final optimisation applied was the activation of asynchronous I/O during the simulation. By default, SeisSol performs I/O as a blocking operation, halting computation until output is written. To mitigate this bottleneck, the `ASYNC` library can be configured to offload I/O to a dedicated thread, allowing I/O and computation to overlap. This reduces I/O-induced stalls and improves overall performance. To enable asynchronous I/O, we set `ASYNC_MODE=THREAD` and also specify the value 8388608 (8 MB) for the `ASYNC_BUFFER_ALIGNMENT`, `XDMFWRITER_ALIGNMENT` and `XDMFWRITER_BLOCK_SIZE` variables. We also set `OMP_NUM_THREADS=63`–reserving one core (i.e. thread) specifically for I/O operations. This tuning led to significant reductions in I/O time by at least 75% on four nodes, which in turn further decreased the average runtime by approximately 50 seconds, as shown in Table 4.2.

Table 4.2: I/O write performance comparison between serial and asynchronous modes for the Türkiye benchmark, run on four nodes of Bridges-2. Each node is configured with 2 MPI ranks and 64 OpenMP threads per rank under the polling communication mode.

Metric	Serial (s)	Async (s)	Change (s)	Percentage (%)	
Time Blocking IO	8.80	0.0113	**-8.79**	**-99.87**	▲
Field Backend	3.84	0.958	**-2.89**	**-75.05**	▲
Field Frontend	3.85	0.0094	**-3.85**	**-99.76**	▲
Fault Backend	14.65	1.585	**-13.07**	**-89.18**	▲
Fault Frontend	14.68	0.0015	**-14.68**	**-99.99**	▲

Final Performance Results

In the final days of the competition, we ran our best-performing configuration–2 MPI ranks with 64 OpenMP threads per rank, polling mode with spread affinity, and asynchronous I/O–on four nodes. Multiple trials were conducted to account for performance variability caused by system noise or node-level load. Our best run, shown in Table 4.3, achieved a peak performance of 3.62 TFLOP/s with a total runtime of 17 minutes and 56 seconds.

Table 4.3: Highest performance obtained for the Türkiye benchmark on Bridges-2 with SeisSol, using 2 MPI ranks per node and 64 OpenMP threads per rank, polling mode with spread affinity, and asynchronous I/O on four CPU nodes.

Metric	Value
Hardware (HW)	3.62 TFLOP/s
Non-zero (NZ)	1.69 TFLOP/s
Simulated Time	17 min 56.85 s (1,076.8 s)
Time in Compute Kernels	2 hours 18 min 58.53 s (8338.53 s)
Load Imbalance	1.93%
Wave Field Writer (Backend)	621.71 ms
Wave Field Writer (Frontend)	8.93 ms
Fault Writer (Backend)	899.67 ms
Fault Writer (Frontend)	1.58 ms

4.5 Task 2: Top Three MPI Calls

Profiling was carried out using two available profilers on Bridges-2–Linaro Map and Scalasca–to identify the most frequently invoked MPI functions throughout the entire runtime. Our analysis revealed that `MPI_Test` was the most frequently called routine, used for background communication progression for non-blocking persistent communications initiated via `MPI_Start`. These calls are made primarily throughout the simulation within two functions:

- `AbstractGhostTimeCluster::testQueue`
- `GhostTimeClusterWithCopy::testReceiveQueue`

where persistent MPI communication is used to exchange halo region data between neighbouring processes for the ghost and copy layers.

However, a discrepancy was observed between the two profiling tools in their reporting of MPI call frequencies. While Scalasca identified both `MPI_Test` and `MPI_Start` as the most frequently invoked functions, Linaro MAP reported only `MPI_Start`. We suspect that Linaro Map may have under-reported, or potentially omitted calls to `MPI_Test` and so we opted to use Scalasca's measurements (shown in Table 4.4).

Table 4.4: A list of the top five most frequently called MPI functions throughout the entire runtime using Scalasca, running the Türkiye benchmark on four nodes on Bridges-2.

Function	Calls	Duration (s)	Bytes Sent	Bytes Received
`MPI_Test`	3.89e10	2.03e04	-	-
`MPI_Start`	3.76e08	827.8	5.41e12	5.41e12
`MPI_Isend`	1.22e07	459.58	1.04e09	-
`MPI_Irecv`	1.22e07	9.22	-	-
`MPI_Wait`	4.96e06	50.75	-	2.93e08

4.6 Task 3: Visualisation of the Türkiye Benchmark

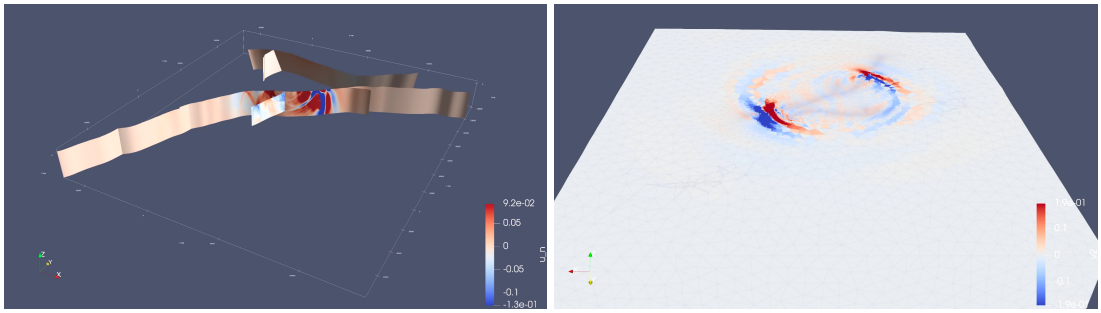
The final task involved running the Türkiye benchmark to generate a short video visualising seismic activity using a tool of choice. For our initial attempt, we used the dedicated input test files provided by the competition committee, specifically prepared for visualisation of the Türkiye scenario. The simulation ran for approximately 5 hours and 12 minutes, producing 1.1GB of binary output data stored in the directory `output_hev1/Turkey_hom_ext4_o6_ev1-fault_cell`.

Upon loading the fault mesh in ParaView, visual inspection confirmed key fault structures, including the East Anatolian Fault (EAF) and the smaller Sürgü–Misis Fault. However, the default configuration only produced three time steps (0, 50, and 150s), resulting in a choppy and limited animation.

To improve the temporal resolution and produce a smoother visualisation, the variable named `printtimeinterval_sec` within the `parameters.par` file was reduced from 50.0 to 1.0 seconds. This adjustment enabled output at every simulation second, totalling 150 frames.

Using the updated configuration, a second simulation was conducted, completing in 5 hours, 50 minutes, and 19 seconds–roughly 38 minutes longer than the original run–and generated 5.5GB of output due to the increased number of snapshots.

Figure 4.5: Türkiye benchmark simulated on Bridges-2 and visualised in ParaView showing fault normal velocity (u_n) at 20 seconds with bilateral, above-surface rupture along the East Anatolian Fault (EAF) from opposing viewpoints: northwest-to-southeast (left) and southeast-to-northwest (right).



When reloaded in ParaView, the higher temporal resolution substantially improved the clarity and fluidity of the animation (Figure 4.5). For analysis of rupture dynamics, the fault normal velocity (u_n) was visualised to highlight compressive zones along the East Anatolian Fault (EAF) (left image). A complementary surface view from above illustrated wave propagation patterns and revealed the underlying non-uniform tetrahedral mesh. This mesh structure used finer elements around the rupture zone to capture high-frequency behaviour, while employing coarser elements further from the fault to reduce computational cost without compromising overall accuracy.

4.7 Challenges Faced during the Competition

Throughout the competition, we encountered several challenges that caused prolonged delays and ultimately reduced the time available for further benchmarking and tuning.

Race Condition When Running Simultaneous Jobs During benchmarking, we encountered a recurring `File Exists` POSIX error originating from the `XDMFWriter` submodule when running multiple SLURM jobs concurrently. The issue stemmed from several SeisSol instances attempting to read from and write to the same `turkey-benchmark` directory simultaneously. In particular, if one instance of SeisSol was writing to the `output_hev1/Turkey_hom_ext4_o6_ev1-fault_cell` file while another tried to access or modify it, a race condition would occur, leading to the crash. To resolve this, we modified the SLURM script to copy the `turkey-benchmark` directory into a unique, job-specific working directory before launching SeisSol. This approach isolated the output of each run and successfully prevented file access conflicts.

Long SLURM Queue Delays due to Hardware Contention One of the most significant challenges encountered during the competition was the prolonged queue times experienced on Bridges-2. Submitted SLURM jobs typically took around three days before execution, but in later stages, queue times extended to a week. This issue was compounded by instances where jobs, after long waits, failed during execution due to script or SeisSol-related errors. Such failures necessitated debugging and resubmission, resulting in additional delays. The command `showuserjobs -a cis240152p`, showed us that approximately 560 nodes were reserved for the `cis240152p` account, a dedicated allocation for the competition. Given the active participation of 26 student teams and researchers sharing this limited resource, high node utilisation became the primary bottleneck leading to these observed queue times.

For example, at the latter stages of the competition, we intended to investigate the impact of node allocation locality by testing SLURM's `–contiguous` option to allocate nodes physically close together. However, due to these extensive queue delays, we were unable to conduct this experiment.

We instead, opted to utilised the interactive nodes on Bridges-2 which were used using the `interact -t 01:00:00 -p RM -N 4 --ntasks-per-node=128` command. Interactive nodes were separate from the main queue and thus did not exhibit the same hardware reservation delays. The team had concerns regarding potential degraded performance; however, our results between the two modes revealed no noticable differences in peak performances.

Bridges-2 System Downtime On Tuesday, 29th April, the Bridges-2 system was shut down due to severe storms in the Pittsburgh area [52] causing a complete power outage. The downtime began at 23:00 UTC and lasted until 20:00 UTC on 5th May. During this time, multiple attempts were made to re-establish a connection to the system, however, these were unsuccessful. As a result of this disruption, benchmarking activities on Bridges-2 had to be temporarily suspended, which further limited our timeline. Due to uncertainty regarding system availability and to mitigate this disruption, we utilised the ARCHER2 system to continue our investigations during the downtime.

Conflicting Timelines The virtual format of the competition started on the 1st April, conflicting with our coursework deadlines, which delayed our active participation until 17th April, at which point the team could begin working on benchmarking and tuning their applications. This delay resulted in the team being unable to conduct a more in-depth performance analysis, which may have allowed more tuning to be conducted.

4.8 Summary and Outcomes

On 12th June, the closing ceremony of the ISC Student Cluster Competition 2025 (ISC SCC25) took place, where the final results for both the in-person and online events were announced. Although the team did not place first, participation in the competition was a great learning experience in porting and tuning a large-scale HPC application such as SeisSol using various tools and techniques.

Whilst substantial effort focused on tunings such as NUMA-aware MPI and thread placement, compiler selection, and the choice of MPI implementation, due to the limited timeframe and the four-node limit, further analysis could not be carried out. As a result, it was not feasible to further explore SeisSol's scaling behaviour at larger node counts or to rigorously evaluate the effectiveness of different optimisation strategies.

Chapter 5

Porting and Performance Analysis on ARCHER2

Building upon the work conducted during the ISC SCC25 competition, this chapter presents a more in-depth analysis of SeisSol's performance on ARCHER2, the UK's national CPU-based supercomputing system. The aim is not only to explore tuning strategies for maximising performance, but also to assess computational efficiency and scalability. Unlike the four-node limit imposed during the competition on Bridges-2, ARCHER2 enables us to use a greater number of compute nodes, allowing for a more comprehensive evaluation of hybrid parallelisation strategies, NUMA-aware configurations, and their impact on performance at scale.

5.1 System Specifications

ARCHER2, supplied by HPE Cray, is operated by EPCC in collaboration with the University of Edinburgh and provided by UK Research and Innovation (UKRI). The platform is an HPE Cray EX system comprising 5,860 CPU compute nodes, including 5,276 standard nodes with 256GB of DDR4 RAM and 584 high-memory nodes with 512GB of RAM [53]. Each node features two AMD EPYC™ 7742 64-core Zen 2 processors operating at 2.00 GHz (downclocked from the nominal 2.25 GHz) [54], including a total of 128 cores per node. The system architecture is modular: each compute blade hosts four nodes, and each cabinet accommodates up to 64 blades, allowing for a maximum of 256 compute nodes per cabinet. With 5,860 nodes in total, ARCHER2 comprises approximately 23 compute cabinets. Inter-node communication is provided by the HPE Cray Slingshot 10 interconnect, delivering 200Gb/s signalling and configured in a dragonfly topology. The system includes two main storage partitions: a 1PB home

filesystem (NetApp FAS8200A) and four 3.6PB work filesystems (HPE Cray L300).

The system runs SUSE Linux Enterprise Server 15 (SLES v15.4 Service Pack 4) and incorporates the HPE Cray Linux Environment (CLE), which provides Cray-specific system tools and configurations tailored for the Shasta architecture. The system also includes the HPE Cray Programming Environment (CPE), offering a suite of compilers and HPC libraries. Three compiler toolchains are available: the AMD Optimising Compiler Collection (AOCC), the GNU Compiler Collection (GCC), and the HPE Cray Compiling Environment (CCE). A single MPI implementation is provided which is HPE Cray MPICH2 v3.4a2. Job submissions and scheduling are managed by SLURM v22.05.11.

5.2 Porting SeisSol to ARCHER2

We begin porting SeisSol onto ARCHER2 by first installing all required dependencies followed by compiling SeisSol itself. All three programming environments were utilised to compare performance across the compiler toolchains with all configurations shown in Table 5.1.

Table 5.1: Configurations detailing the compiler toolchain, MPI implementation and dependency versions used for for SeisSol v1.3.1 on ARCHER2.

Components	Configuration 1	Configuration 2	Configuration 3
Build Toolchains			
Compiler	GCC v11.2.0 20210728	AOCC v4.0.0 (LLVM 14.0.6)	Cray Clang v15.0.0
MPI	Cray MPICH v3.4a2	Cray MPICH v3.4a2	Cray MPICH v3.4a2
Dependencies			
SeisSol	1.3.1	1.3.1	1.3.1
HDF5	1.12.2	1.12.2	1.12.2
NetCDF	4.9.0	4.9.0	4.9.0
yaml-cpp	0.6.3	0.6.3	0.6.3
Lua	5.4.4	5.4.4	5.4.4
EASI	1.0	1.0	1.0
ParMETIS	4.0.3	4.0.3	4.0.3
Eigen	3.4.0	3.4.0	3.4.0
LIBXSMM	1.17	1.17	1.17
PSpaMM	1.0	1.0	1.0

Given the number of dependencies and the variety of compilers, versions, and build configurations required, each dependency was compiled individually using a dedicated installation script. To maintain consistency, all dependencies within a given compiler

toolchain were built using the same compiler. Although the compilation process was non-trivial, no major issues were encountered.

With all required dependencies successfully compiled, we next built SeisSol with the same configuration options as used on Bridges-2. We compiled SeisSol v1.3.1 in `Release` mode that enables `-O3` and set host architecture to `rome` which selects the `-march=znver2` and `-mtune=znver2` flags. For the simulation, we set the precision to `double`, the equations was set to `elastic` (i.e., isotropic elastic material) and the convergence order to four. The graph partitioning library was set to `parmetis` and the kernel generator was set to one of three options per build: `Eigen`, `LIBXSMM` or `PSpaMM`.

Throughout the process of recompiling SeisSol for different compilers and kernel generators, we would occasionally encounter Python runtime errors during the YaTeTo "Calling external code generators..." CMake stage. This was due to files conflicting with previously generated code by other builds. Therefore, to ensure that we avoid potential conflicts, we specifically delete the `build/src/generated_code` directory (if it exists) before invoking CMake so that we have a clean build folder.

5.2.1 Evaluating Idealised Single-Node Performance

We first used the proxy to test that all kernels ran without errors, and also evaluated which compiler/generator configuration obtains the highest idealised single-node performance excluding I/O and communication. The proxy was executed using a single MPI process and 128 OpenMP threads, fully utilising a node. Threads were bound to physical cores via `OMP_PLACES=cores` and assigned using a close binding policy `OMP_PROC_BIND=close`, which allocates 64 threads to the first socket before filling the second. Hyperthreading was disabled, and a block distribution strategy was used (`--hint=nomultithread --distribution=block:block`) ensuring that the MPI process and its OpenMP threads were placed on one socket before moving to the next. The `SRUN_CPUS_PER_TASK=$SLURM_CPUS_PER_TASK` variable was explicitly set to prevent incorrect resource binding by SLURM. The simulation was run for 100 timesteps across all computational kernels (`all`), with the total number of cells ranging from 512 to 524,288–doubling at each step–to evaluate the effect of problem size on computational throughput and the peak performance achievable on a single node.

Figure 5.1: Idealised single-node performance across nine compiler-kernel generator configurations using the SeisSol proxy (excluding MPI communication and I/O) on ARCHER2. The benchmarks were run between 512 and 524288 elements for 100 timesteps, testing all kernels.

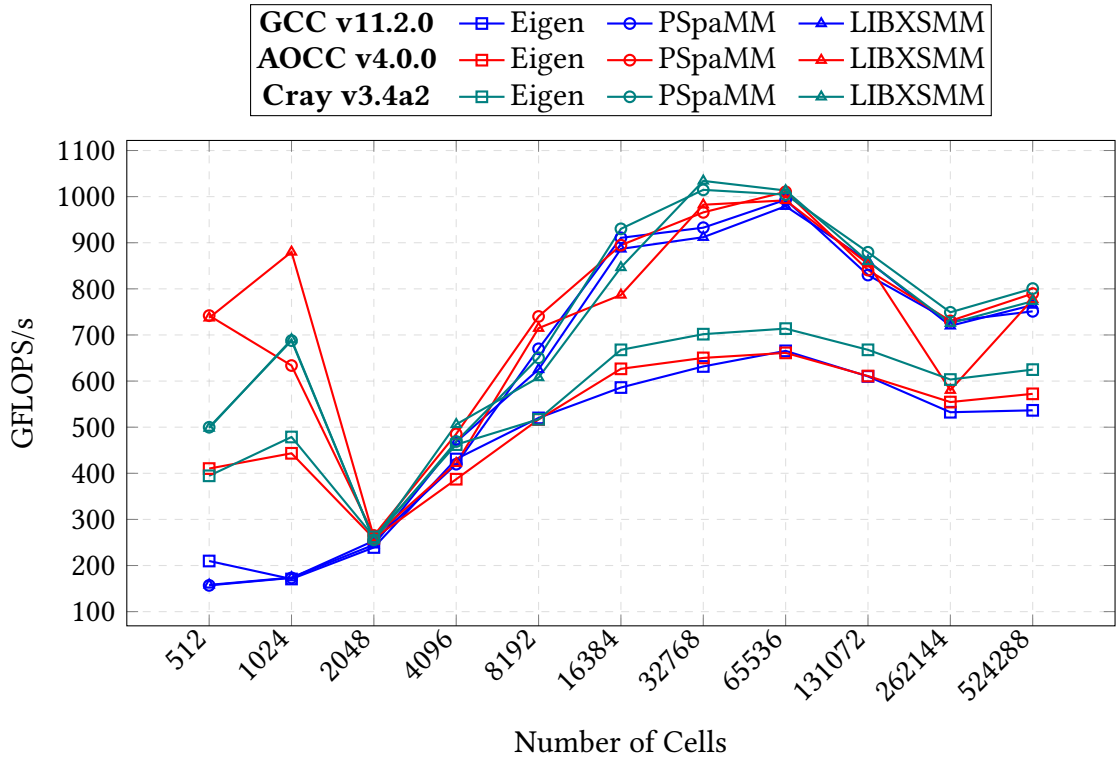


Figure 5.1 presents results for nine combinations of compiler and code generator. Across all three compiler toolchains, a clear trend emerges: both LIBXSMM and PSpaMM deliver substantially higher performance than Eigen.

For problem sizes below 2,048 cells, performance diverges, with the Cray and AOCC compilers outperforming GCC. However, these small-scale cases are not representative of the large-scale simulations of practical interest and, as such, have limited importance when determining the most suitable compiler–generator combination.

While absolute performance varied–with a maximum standard deviation of approximately 185 GFLOP/s between generators–all three compilers exhibited the same general trend: performance increased with problem size, peaking between 32,768 and 65,536 cells at roughly 950 GFLOP/s, before declining. We suspect this drop is due to the problem size exceeding the 16 MB L3 cache capacity of the AMD EPYC™ 7742, leading to cache evictions and increased memory access latency.

At 65,536 cells, the three compilers for both LIBXSMM and PSpaMM achieved their closest and highest performances, differing by only 34 GFLOP/s. The highest performance achieved was with the Cray compiler and LIBXSMM generator, achieving 1034.09 GFLOP/s. While the Cray compiler consistently produced slightly higher FLOP rates than AOCC or GCC, GCC with the LIBXSMM generator was ultimately chosen for three reasons. First, although GCC did not deliver the absolute peak performance, it produces comparable results while maintaining consistency with the ISC SCC25 setup on Bridges-2, enabling more meaningful cross-system comparisons.

Second, LIBXSMM demonstrated marginally better consistency across problem sizes, with a lower standard deviation (308.27 vs. 313.95) and coefficient of variation (49.92% vs. 50.63%) compared to PSpaMM. More importantly, LIBXSMM is optimised for the AVX2 instruction set, which matches the Zen 2 architecture on ARCHER2, whereas PSpaMM targets AVX-512, which is not natively supported on Zen 2 and would therefore be less efficient. Choosing LIBXSMM ensured that the generated code uses AVX2-optimised instructions suited to the hardware.

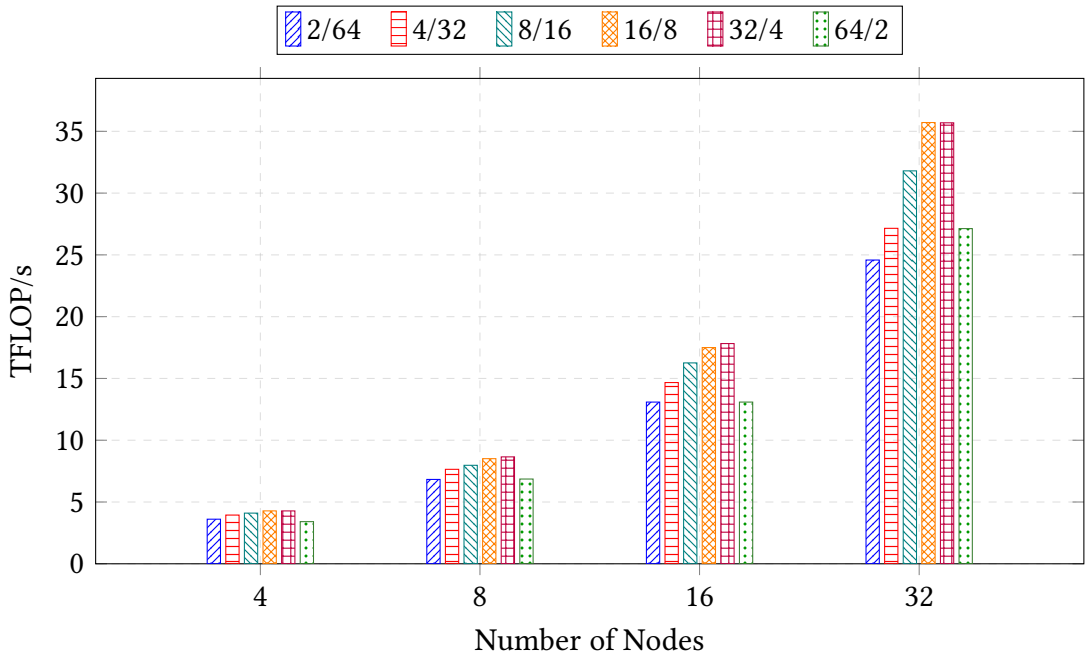
Finally, previous studies have shown LIBXSMM to outperform PSpaMM in large-scale SeisSol simulations beyond the scope of the proxy benchmark [55].

5.3 Hybrid Parallelism and Strong Scaling Evaluation

Using the GCC compiler and LIBXSMM code generator, we next evaluated which MPI and OpenMP configurations deliver the best performance for the Türkiye benchmark, taking into account ARCHER2's NUMA topology. To ensure reasonable runtimes while testing multiple configurations, the `EndTime` parameter (i.e., the maximum simulated time) was reduced from 10 to 5 seconds. All runs were performed with polling-based communication enabled.

Running `numactl -H` revealed that each ARCHER2 node consists of eight NUMA regions–four per socket across the node's two sockets. Each region contains 16 physical cores (organised as two CCDs) and is paired with 64GB of DDR4 memory, corresponding to an `NPS=4` configuration [56].

Figure 5.2: Strong scaling of the simulation (excluding initialisation) across varying node counts for the Türkiye scenario, comparing six MPI/OpenMP configurations under the polling communication mode on ARCHER2.

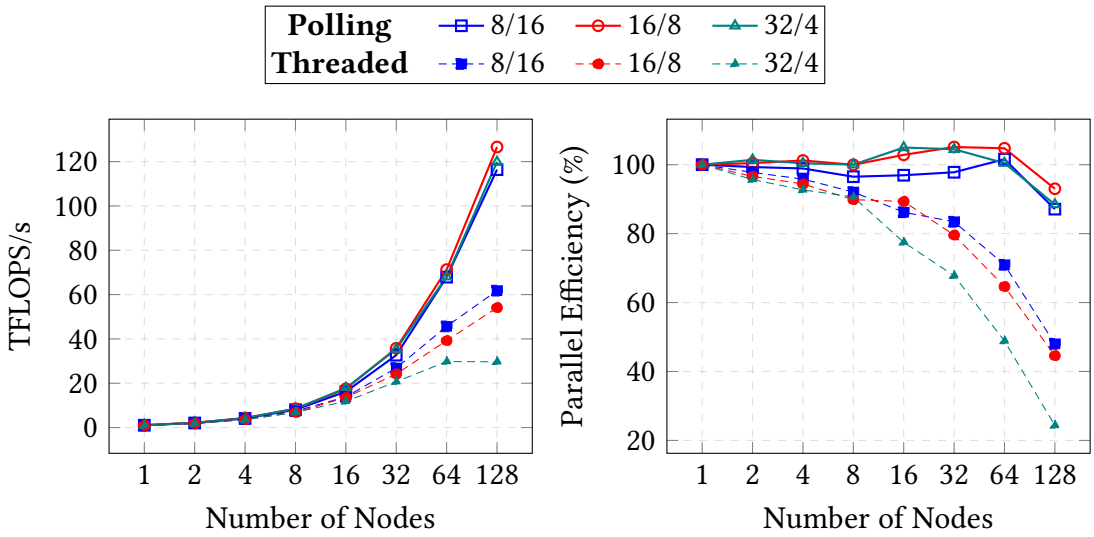


The results in Figure 5.2 show that the optimal MPI/OpenMP configuration varies with the node count. At 32 nodes, the 16/8 and 32/4 configurations achieve the highest performance, reaching 35.7 and 35.6 TFLOP/s, respectively. The 8/16 setup is the next best, achieving 31.7 TFLOP/s–around 4 TFLOP/s lower than 16/8. This indicates that running with at least eight MPI ranks, aligned with the underlying NUMA layout and balancing MPI and OpenMP, performs better than extreme configurations such as 2/64 and 64/2.

On ARCHER2's AMD EPYC 7742 nodes, these configurations correspond to different binding granularities. With eight MPI ranks per node, each rank is bound to an entire NUMA region comprising two CCDs (16 cores). With 16 MPI ranks, each rank is bound to a single CCD (8 cores). With 32 MPI ranks, each rank occupies half a CCD (4 cores, equivalent to a compute complex), maximising locality but increasing the number of MPI processes.

Using these three best-performing configurations, we next assessed the full range of nodes from 1 to 128 nodes, analysing both the main simulation phase and the impact of enabling a dedicated communication thread via the `SEISSOL_COMMTHREAD` environment variable.

Figure 5.3: Strong scaling and parallel efficiency of the simulation (excluding initialisation) from 1 to 128 nodes for the Türkiye scenario, comparing three MPI/OpenMP configurations, as well as polling and threaded communication modes on ARCHER2.



Our strong scaling results, shown in Figure 5.3 (left), demonstrate that SeisSol scales efficiently under polling-based communication, achieving peak performances of 116.38, 126.59, and 119.84 TFLOP/s for the 8/16, 16/8 and 32/4 configurations, respectively, at 128 nodes. The best-performing configuration, 16/8, attained 126.59 TFLOP/s–equivalent to 21.14% of the system's theoretical peak performance. At 64 nodes, the 16/8 setup outperforms 32/4 by approximately 4.7%, with the gap widening to 5.6% at 128 nodes. This indicates that while the configurations perform comparably at smaller scales, performance differences become more pronounced as node count increases.

Using a communication thread, on the other hand, leads to substantially lower performance of 61.78, 54.11, and 29.68 TFLOP/s for the same configurations. Furthermore, we can see a clear inverse relationship between the number of MPI ranks and scalability when the communication thread is enabled–suggesting that increasing the number of MPI ranks more negatively impacts performance under thread-based communication.

The parallel efficiency (right) for all three configurations remains at a high level with a linear speedup up to 8 nodes for the 16/8 and 32/4 configurations. Between 16 and 64 nodes, we observe superlinear speedup for both configurations with 16/8 not only achieving the highest TFLOP/s but also sustains the highest efficiency, reaching a peak of 104% at 64 nodes.

Beyond this point, parallel efficiency begins to decline–dropping to 93% for 16/8, 88% for 32/4, and 87% for 8/16 at 128 nodes. The strong performance and efficiency ob-

served at 64 nodes is consistent with the proxy application results shown in Figure 5.1. In the full Türkiye scenario, which comprises 2,410,756 cells, distributing the workload across 64 nodes results in approximately 37,668 cells per node–closely aligning with the cell counts that achieved optimal single-node performance in the proxy. It is important to note, however, that this per-node cell count is an approximation. In practice, the exact distribution may vary slightly due to SeisSol's local time stepping strategy and the weights that are assigned to each node, which affect how the mesh is distributed. Nonetheless, the alignment between the estimated per-node workload and the proxy's optimal range supports the conclusion that 64 nodes represent a locally optimal configuration for this scenario.

When a dedicated communication thread is enabled, parallel efficiency trends closely mirror the TFLOP/s results, consistently yielding significantly lower values. With each doubling of the node count, efficiency declines progressively–reaching just 55%, 48%, and 28% of the corresponding polling-mode efficiency at 128 nodes for the 8/16, 16/8, and 32/4 configurations, respectively. This highlights a clear scalability bottleneck introduced by the communication thread used to progress asynchronous communication, which becomes increasingly pronounced at larger scales.

Given the significant performance disparity between polling and threaded communication, we investigated potential tunings that can be applied to improve the performance of the threaded configuration. During initialisation, SeisSol prints out both the OpenMP worker and communication affinity, which by default, assigns the communication thread for each MPI rank to multiple logical cores and/or physical cores. This irregular placement was suspected to be a potential performance bottleneck as memory accesses between threads would cross over multiple NUMA nodes, resulting in increased latency.

To mitigate this, SeisSol provides the `SEISSOL_FREE_CPUS_MASK` environment variable, which allows users to explicitly define a set of CPU cores to be reserved for communication/IO. This ensures a predictable and efficient CPU binding for each MPI rank. The i^{th} entry in the list corresponds to the core reserved for the i^{th} MPI rank. Listing 5.1 illustrates how to configure this variable when running with 16 MPI ranks per node, reserving the last physical core of each rank for its communication thread and/or asynchronous IO.

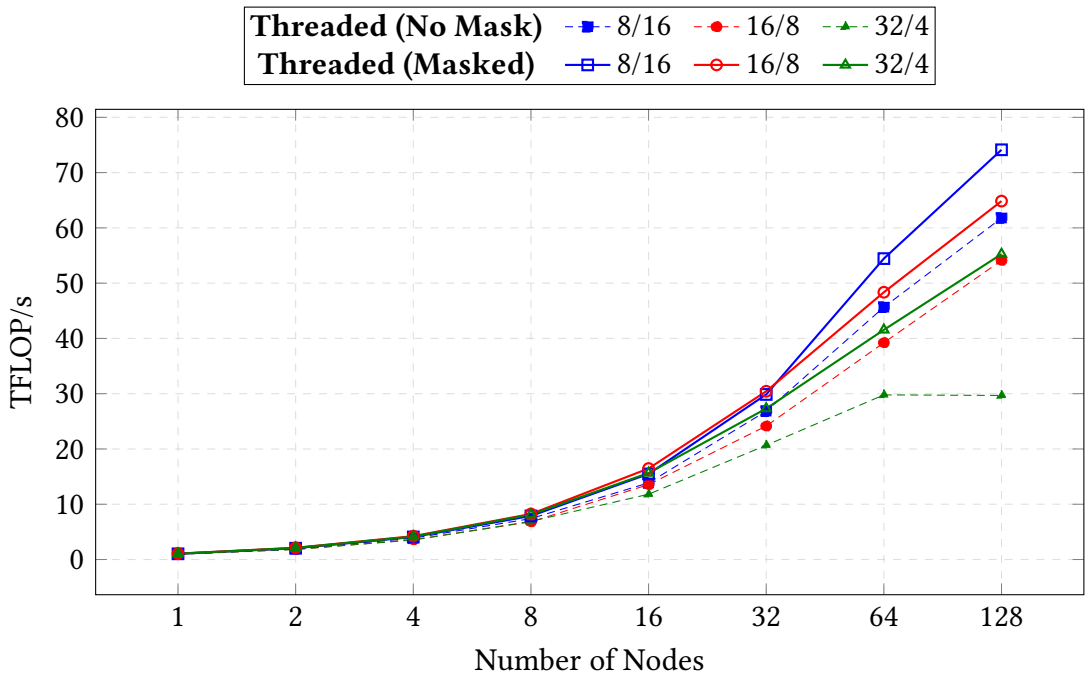
Listing 5.1: Example CPU affinity mask configuration for SeisSol using 16 MPI ranks per node and 8 OpenMP threads per rank, reserving the last core of each rank for communication and I/O threads.

```
export SEISSOL_FREE_CPUS_MASK="7,15,23,31,39,47,55,63,71,79,87,95,103,111,119,127"
```

With this variable set, we reran the simulation from 1 to 128 nodes and compared it against not explicitly setting a mask. As illustrated in Figure 5.4, all hybrid con-

figurations showed notable performance improvements when the mask was applied. This indicated that explicitly binding communication threads to dedicated cores enhances resource utilisation and reduces contention, leading to better overall performance. Specifically, we observed speedups of 19%, 23%, and 39% for the 8/16, 16/8, and 32/4 configurations, respectively. Importantly, however, this tuning did not address the significantly worse performance compared to polling.

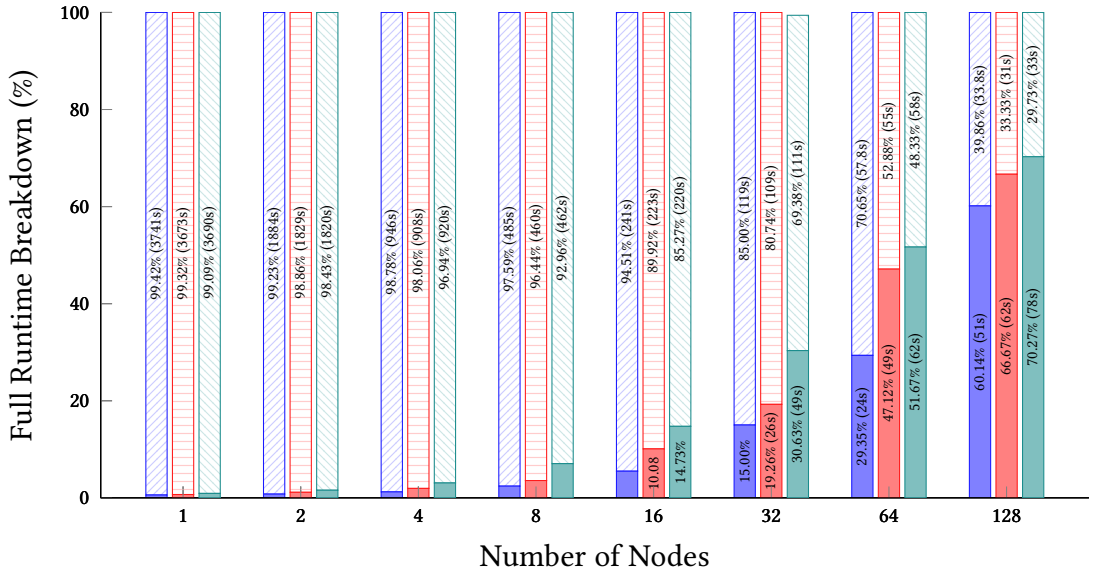
Figure 5.4: Strong scaling of the Türkiye benchmark in threaded communication mode, comparing runs with and without a CPU affinity mask, using 16 MPI ranks per node and 8 OpenMP threads per rank, across 1 to 128 nodes on ARCHER2.



5.4 Analysing Startup Performance Bottlenecks

Although the 16 MPI ranks per node and 8 OpenMP threads per rank configuration achieved the best simulation scalability, analysis of SeisSol's timing outputs at 64 and 128 nodes revealed that a substantial portion of the total runtime (excluding simulation) was spent in initialisation. This prompted further investigation into non-simulation factors–such as initialisation, I/O, and MPI communication setup–that can significantly impact scalability at large node counts. Across the three hybrid configurations, we analysed the breakdown of total runtime from 1 to 128 nodes.

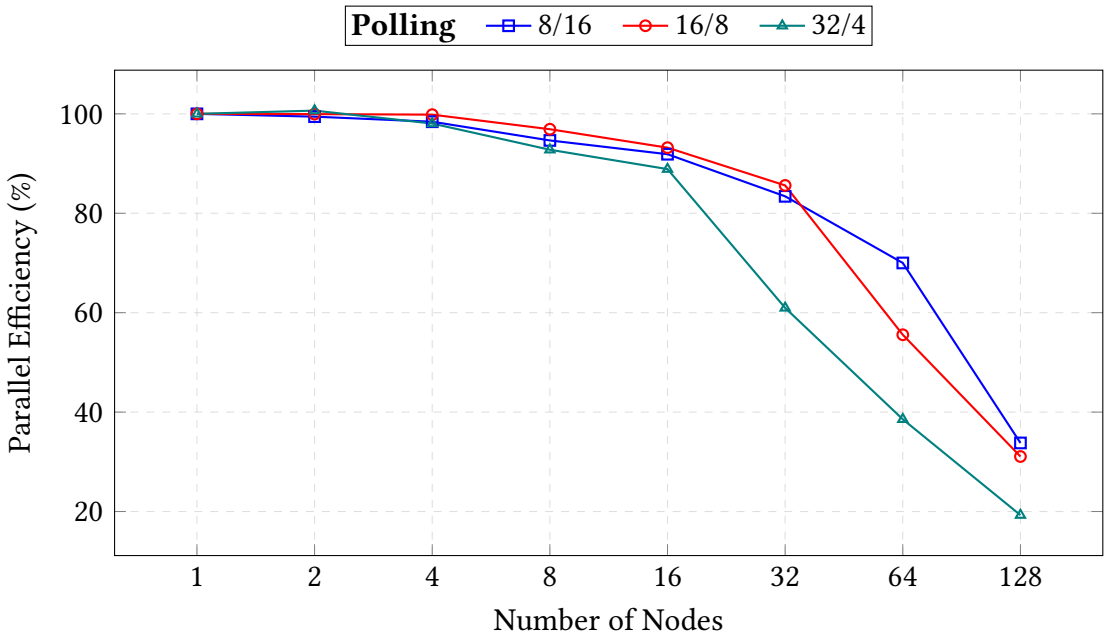
Figure 5.5: Full runtime breakdown of the Türkiye benchmark for the polling communication mode, comparing three MPI/OpenMP configurations. Bars show the relative contributions of initialisation and simulation times for runs from 1 to 128 nodes on ARCHER2.



The timing breakdown shown in Figure 5.5 illustrates a clear trend. From 1 to 16 nodes, the simulation phase dominates, with initialisation accounting for less than 20% of total runtime. However, while initialisation time initially decreases with increased node count, it begins to rise again beyond 16 nodes. At 64 nodes, initialisation comprises nearly 50% of the total runtime for the 16/8 and 32/4 configurations, and at 128 nodes, it reaches approximately 65% across all configurations.

This trend highlights that although the simulation phase scales efficiently (as shown previously in Figure 5.3) and decreases with node count, the growing cost of initialisation increasingly offsets these gains. As a result, improvements in total runtime become increasingly constrained, not due to simulation inefficiency, but due to the disproportionately rising cost of setup for greater number of nodes. Furthermore, it indicates that the more MPI ranks are used the longer the initialisation takes.

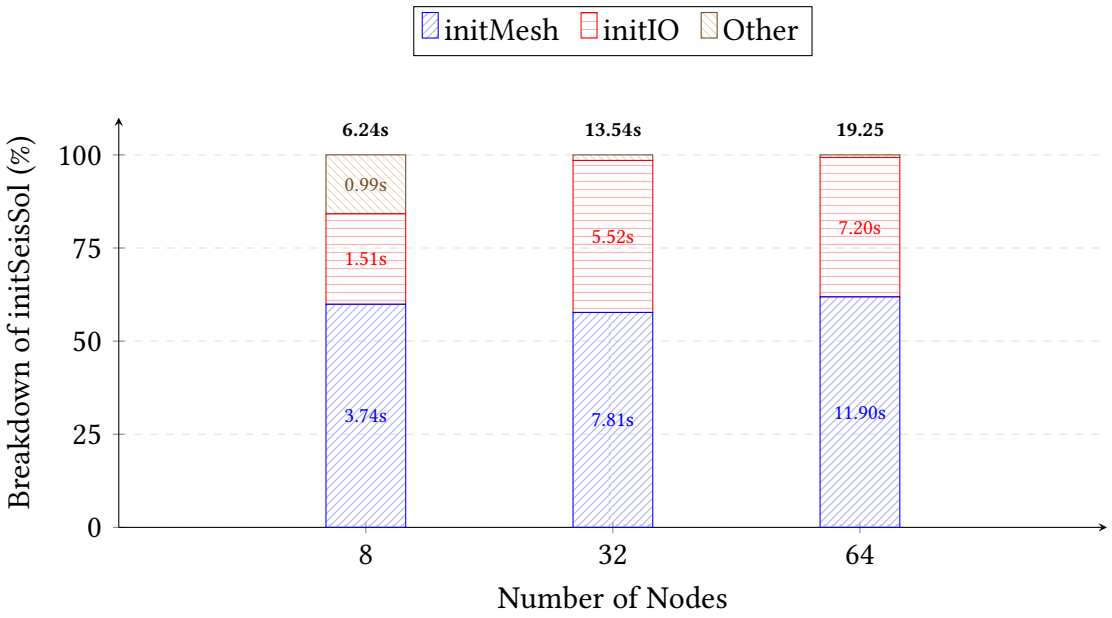
Figure 5.6: Parallel efficiency of the full runtime (incuding initialisation and simulation) from 1 to 128 nodes for the Türkiye benchmark comparing three MPI/OpenMP configurations for the polling communication mode on ARCHER2.



This is reflected in the full runtime parallel efficiency (Figure 5.6) which confirms that efficiency declines more sharply with increasing node count, largely influenced by the number of MPI ranks. Efficiency drops beyond four nodes for all configurations, but at progressively lower node counts as rank count increases–starting at 16 nodes for 32 ranks, 32 nodes for 16 ranks, and only beyond 64 nodes for 8 ranks. If total runtime were dominated by the simulation, both simulation and full runtime efficiency curves would align; instead, the divergence shows how initialisation limits end-to-end performance.

While the 8/16 configuration achieved the shortest initialisation time, the 16/8 setup consistently delivered the highest simulation performance. To balance both objectives–maximising simulation throughput and mitigating the initialisation bottleneck–the 16/8 configuration was chosen as it offered the lowest overall runtime when considering both phases together. This configuration was therefore selected for a focused analysis of initialisation behaviour. Profiling was conducted using CrayPat v22.12.0 at 8, 32, and 64 nodes to examine how performance bottlenecks evolved with increasing scale.

Figure 5.7: Breakdown percentage of the `initSeisSol` function on 8, 32 and 64 nodes using the CrayPat v22.12.0 profiler running 16 MPI ranks per node and 8 OpenMP threads per rank for the the Türkiye benchmark in polling communication mode on ARCHER2.



The runtime callstack was analysed for each of the node counts to understand where the majority of the time was being spent and how it may potentially change as we scale up. The main function called during initialisation is `initSeisSol` which primarly contains calls to four functions, namely, `initMesh`, `initModel`, `initSideConditions` and `initIO`. The `initMesh` and `initIO` functions together account for over 80% of the initialisation time at 8 nodes, increasing to over 98% beyond 32 nodes as shown in Figure 5.7. The time spent in both functions increases proportionally with the overall initialisation time, with `initMesh` consistently accounting for around 60%.

A detailed examination of the *callers*, *calltree*, and *profile* reports generated by the `pat_report` tool identified MPI collective communication as the main factor driving the prolonged initialisation times in both functions. In particular, `MPI_Alltoallv` is responsible for 30.1%, 74.3% and 79.0% of the initialisation time at 8, 32 and 64, respectively, indicating a potential underlying performance bottleneck in the MPI implementation. These calls originate from the PUML `generate_mesh` and `partition` functions, which perform domain decomposition to distribute the mesh across MPI ranks. Calls also originate from within the wave field and fault writers (see Table B.5 in the appendix).

5.5 Improving Startup via MPI Backend Tuning

There are two primary communication frameworks supported on ARCHER2: OFI and UCX. Recent investigations by EPCC analysed the scalability of both communication libraries on ARCHER2 and reported varying performance characteristics across applications, particularly in the presence of MPI collective communication [57, 58]. Similarly, the Pawsey Supercomputing Research Centre (PSRC) identified scalability limitations in `libfabric` versions prior to 1.15, especially for applications with substantial point-to-point traffic [59]. Both studies ultimately recommend enabling eager connection establishment, achieved with the environment variables shown in Listing 5.2. These have shown to improve MPI communication performance under OFI especially for collective communication.

Listing 5.2: Enabling eager connection startup for Cray MPICH v3.4a2 which establishes connections between MPI ranks during initialisation instead of at first communication.

```
export MPICH_OFI_STARTUP_CONNECT=1
export MPICH_OFI_RMA_STARTUP_CONNECT=1
```

By default, `libfabric` establishes connections lazily–connections between endpoints are deferred until the first actual communication occurs. While this approach reduces memory overhead and improves startup times, particularly for large-scale runs [60], it can lead to substantial runtime overhead for applications like SeisSol. The use of `MPI_Alltoallv` during mesh partitioning and I/O initialisation introduces all-to-all communication patterns with non-uniform message sizes. Given the $O(P^2)$ communication complexity and large number of process counts involved, lazy connection setup leads to degraded performance during these early phases as a result of a potential sub-optimal collective implementation.

ARCHER2 uses OFI `libfabric` version 1.12 and disables eager connection setup by default. To address this, the `MPICH_OFI_STARTUP_CONNECT` environment variable was enabled to force eager establishment of communication during `MPI_Init_thread`. This approach preemptively sets up connections between MPI ranks, thereby eliminating the cost of runtime connection setup–particularly impactful for collectives such as `MPI_Alltoallv`.

Additionally, we also evaluate the performance impact of switching to UCX v1.9.0 by loading the `craype-network-ucx` and `cray-mpich-ucx` modules which replaces OFI with UCX as the active backend for the MPI communication library.

After performing our strong scaling benchmarks, we profiled the time spent in MPI function calls across 8, 32, and 64 nodes, comparing three configurations: the default OFI backend, OFI with eager connection setup (startup flags), and UCX.

Table 5.2: MPI function runtimes in seconds as reported by the CrayPat v22.12.0 profiler showing the relative differences for 8, 32, and 64 nodes using OFI, OFI + Flags, and UCX backends. The runs were performed for the Türkiye benchmark in polling communication mode on ARCHER2.

Nodes	MPI Function	OFI (s)	OFI+Flags (s)	UCX (s)	OFI+Flags Diff (%)	UCX Rel. Diff (%)
8	MPI_Test	30.873	52.867	33.213	71.24	7.58
	MPI_Start	2.392	3.657	1.441	52.89	-39.75
	MPI_Alltoallv	1.880	0.056	0.052	-96.98	-97.22
	MPI_Comm_split	0.474	0.491	0.439	3.54	-7.34
	MPI_Waitall	0.384	0.417	0.527	8.70	37.26
	MPI_Comm_split_type	0.199	0.231	0.198	16.37	-0.31
	MPI_Alltoall	0.112	0.001	0.014	-98.43	-87.49
	MPI_Isend	0.005	0.007	0.004	32.52	-25.45
	MPI_Scan	0.003	0.003	0.006	11.31	87.66
	MPI_Allreduce	0.002	0.002	0.002	3.78	-9.28
32	MPI_Test	22.106	40.655	19.267	83.90	-12.84
	MPI_Alltoallv	10.063	0.083	0.054	-99.17	-99.46
	MPI_Start	1.875	2.923	1.256	55.86	-33.00
	MPI_Comm_split	0.577	0.413	0.424	-28.48	-26.47
	MPI_Comm_split_type	0.244	0.206	0.205	-15.75	-15.93
	MPI_Alltoall	0.209	0.001	0.004	-99.16	-97.75
	MPI_Waitall	0.106	0.102	0.187	-3.72	76.12
	MPI_Isend	0.039	0.007	0.003	-79.94	-89.85
	MPI_Barrier	0.032	0.010	0.004	-65.86	-85.53
	MPI_Allreduce	0.004	0.001	0.036	-68.46	761.97
64	MPI_Test	16.570	37.597	15.842	126.89	-4.40
	MPI_Alltoallv	15.210	0.082	0.068	-99.46	-99.55
	MPI_Start	1.394	2.406	1.008	72.63	-27.70
	MPI_Comm_split	0.778	0.432	0.425	-44.41	-45.34
	MPI_Alltoall	0.237	0.002	0.009	-98.87	-95.82
	MPI_Comm_split_type	0.216	0.205	0.209	-5.34	-3.54
	MPI_Isend	0.116	0.008	0.003	-92.51	-96.65
	MPI_Waitall	0.087	0.091	0.149	4.42	70.57
	MPI_Barrier	0.062	0.008	0.011	-86.12	-81.16
	MPI_Bcast	0.008	0.000	0.000	-97.83	-94.97

Table 5.2 highlights performance changes relative to the default OFI baseline, with green cells indicating improvements and red cells showing regressions. The most significant gain comes from enabling OFI with startup connection flags, which drastically reduces the runtime of the `MPI_Alltoallv` collective by over 96%, cutting execution times from 1.80, 10.06, and 15.21 seconds to just 56, 83, and 82 milliseconds at 8, 32, and 64 nodes, respectively.

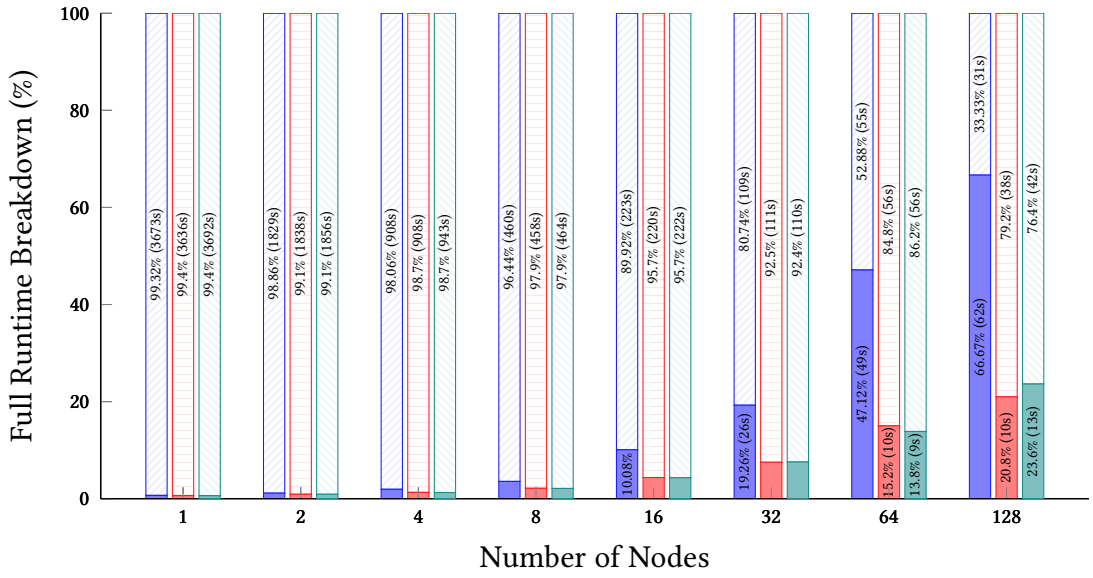
Interestingly, running on 8 nodes, MPI calltimes increased with the most pronounced slowdowns seen in `MPI_Test` and `MPI_Start`. At larger scales of 32 and 64 nodes, performance improvements become more widespread, with eight of the top ten functions showing runtime reductions relative to the default OFI configuration. Despite this overall positive trend, `MPI_Test` and `MPI_Start` remain exceptions: at 64 nodes, their runtimes increase by 21 and 2.1 seconds, corresponding to slowdowns of 126%

and 72%, respectively.

For the alternative communication library, UCX, these collective communication bottlenecks are mostly avoided and consistently outperforms OFI with startup flags across all node counts. Moreover, UCX does not exhibit the same slowdowns for `MPI_Test` and `MPI_Start`.

These results show that MPI call times are consistently lowest when using the UCX communication layer, indicating that its implementation and/or default tuning is more advantageous. This confirms that opting for the non-default UCX library on ARCHER2 leads to the most substantial reduction in communication overhead compared to both OFI configurations.

Figure 5.8: Full runtime breakdown of the Türkiye benchmark using polling communication mode, comparing OFI with startup flags and UCX against the default OFI configuration. All runs use 16 MPI ranks per node and 8 OpenMP threads per rank. Bars show the relative contributions of initialisation and simulation times across runs from 1 to 128 nodes on ARCHER2.



We reassess the significance of these MPI-level improvements on the overall impact for the full runtime (Figure 5.8). The results confirm a substantial reduction in initialisation time, particularly beyond 8 nodes. At 64 nodes, using OFI with startup flags reduces initialisation time by 79.6%, while switching to UCX achieves an even greater reduction of 81.6%. Running UCX on 64 nodes achieves the lowest initialisation overhead taking just 9 seconds.

Figure 5.9: Full runtime parallel efficiency of the Türkiye benchmark using polling communication mode, comparing OFI with startup flags and UCX against the default OFI configuration. All runs use 16 MPI ranks per node and 8 OpenMP threads per rank from 1 to 128 nodes on ARCHER2.

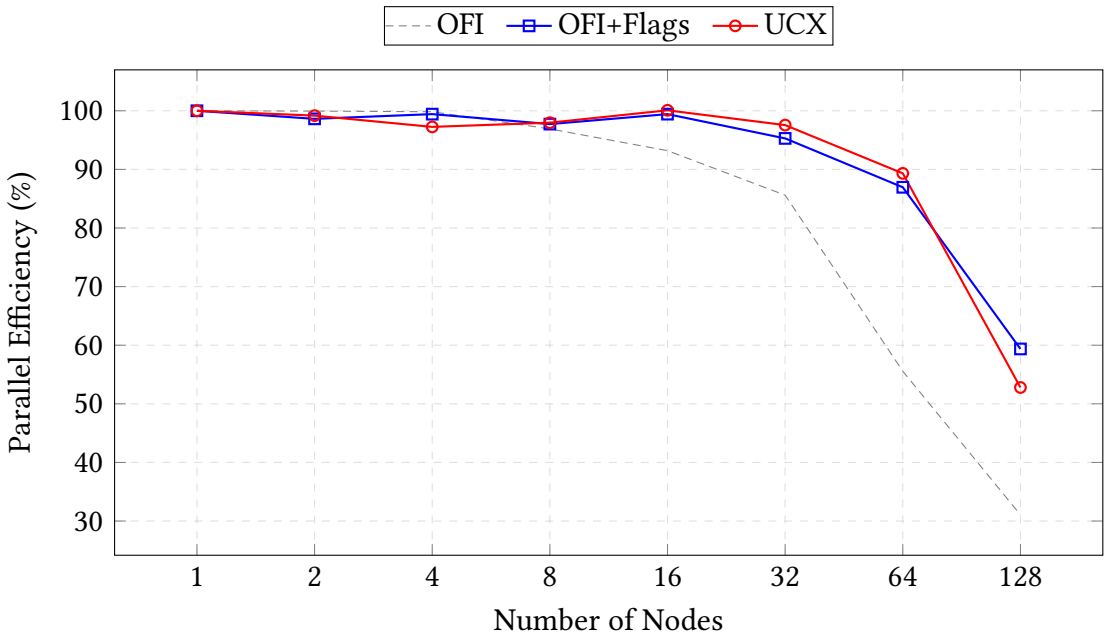


Figure 5.9 demonstrates how reductions in initialisation time lead to notable gains in parallel efficiency. Between 16 and 64 nodes, average efficiency increases by 24.81% with OFI and startup connection flags, and by 27.37% when using UCX compared to the baseline OFI. Notably, the UCX configuration achieves peak efficiency of 100% at 16 nodes, with a total runtime of 232 seconds. Scaling to 64 nodes reduces efficiency slightly to 89%–still a highly efficient value–while significantly lowering the total runtime to just 64.9 seconds (the fastest time-to-solution). These confirm that UCX provides the best overall runtime efficiency, enabling more effective utilisation of available compute resources.

5.6 Summary and Outcomes

Our porting and performance analysis on ARCHER2 has produced several important insights into the optimal configuration and communication strategies for running SeisSol efficiently. Through systematic benchmarking and communication tuning, we identified the following key findings:

- **Polling-based communication offers superior performance over threaded communication** which is consistent with the findings reported in the *D1.2 Performance Investigation* of the ENERXICO project. While earlier work–optimised for Intel Knights Corner/Landing architectures–found that using a dedicated communication thread could improve performance [29, 34], these architectures exhibited negligible NUMA effects. In contrast, our results show that polling-based communication outperforms threaded communication. Similar to ENERXICO's conclusions, we attribute this to the stronger NUMA characteristics of the Zen 2 architecture, where achieving optimal locality requires higher MPI rank counts, making the cost of reserving cores for communication more significant [55].
- **16 MPI ranks per node with 8 OpenMP threads per rank is the most optimial hybrid configuration**, delivering the highest flop-rate at 64 nodes and highest parallel efficiency at 16 nodes for the Türkiye benchmark.
- **Non-default UCX communication library reduces MPI bottlencks** for collective communication. In particular, `MPI_Alltoallv` call durations were reduced by over 90%, significantly reducing initialisation times and contributing to shorter overall runtimes with improved parallel efficiency.

The choice of node count ultimately depends on user priorities–specifically, whether the aim is to minimise computational cost or reduce wall-clock time. If the primary objective is to maximise computational efficiency and minimise resource usage, then running on 16 nodes with UCX is the most economical option. However, for users prioritising faster turnaround and lower time-to-solution–particularly for large-scale studies or time-sensitive workloads–using 64 nodes with UCX strikes the best balance between runtime performance and cost on ARCHER2.

To illustrate the practical impact of these tunings, consider running the Türkiye benchmark 100 times. Using the default ARCHER2 configuration with OFI v1.12 would consume 185 compute units (CUs). In contrast, switching to UCX v1.9.0 reduces this to 116 CUs–a saving of 69 CUs. Such reductions would translate into substantial cost savings, particularly for larger simulations or repeated runs.

5.6.1 Performance Comparison of Bridges-2 and ARCHER2

We now present a direct comparison between the CPU-based systems Bridges-2 and ARCHER2. Although both systems use the Zen 2 processor architecture, SeisSol's performance was affected differently by the choice of communication mode. On Bridges-2, switching between polling and threaded communication resulted in negligible performance differences, in contrast to the clear divergence observed on ARCHER2. We suspect that this disparity can be attributed to two key factors.

First, the number of NUMA regions per node has a direct impact on the performance of threaded communication as previously outlined in our findings. Although enabling a dedicated communication thread with core masking yields better results than omitting the mask (Figure 5.4), it still underperforms compared to the polling mode. The primary reason is the reduction in computational resources: with a communication thread, each MPI rank must reserve one core for communication, thereby decreasing the number of cores available for actual computation.

In SeisSol, the number of MPI ranks must be at least equal to the number of NUMA regions to ensure memory locality. This implies that at least one core per node (0.78% of a ARCHER2's node) is reserved for communication. Scaling up to 32 MPI ranks would reserve 32 cores–amounting to 25% of the node. On systems with many NUMA regions, such as ARCHER2, increasing the number of MPI ranks helps maintain locality and reduce costly cross-region memory accesses. However, under the threaded communication model, this also means a larger fraction of cores are reserved for communication tasks.

This explains why configurations like 32/4 perform worse than 16/8, and 16/8 worse than 8/16: despite better locality, the growing overhead from core reservation outweighs the benefits.

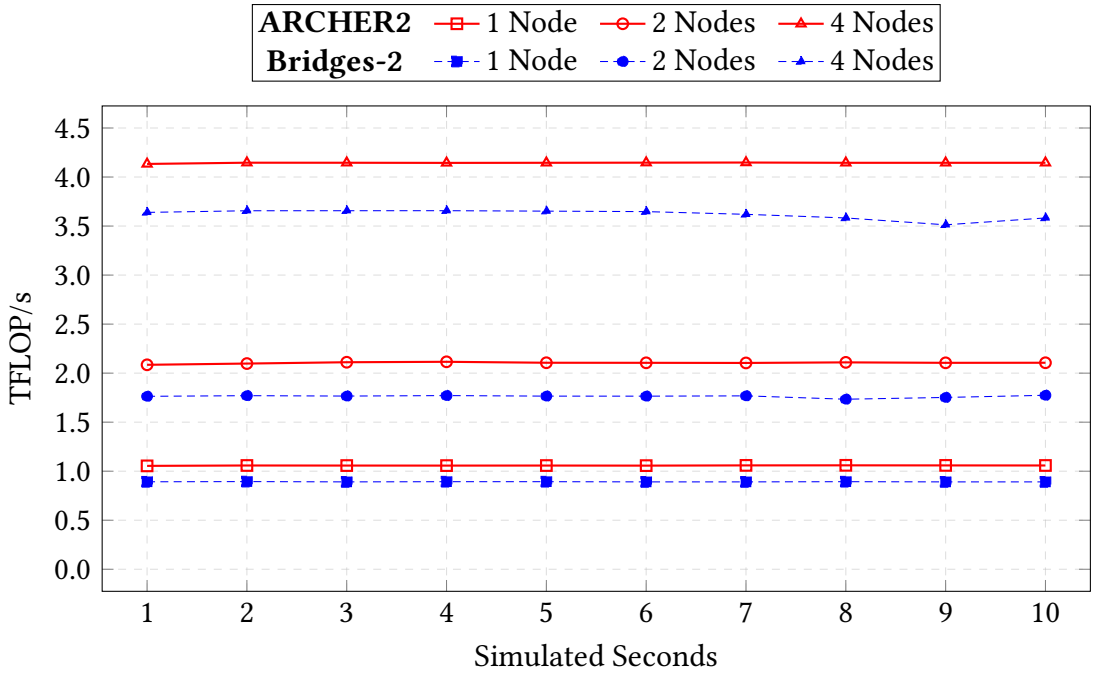
In contrast, Bridges-2 nodes comprise only two NUMA regions, leading to an optimal configuration of just two MPI ranks per node–substantially fewer than the 16 ranks per node required for NUMA-aware placement on ARCHER2. Therefore, only 2 out of 128 cores (1.56%) were withheld from computation on Bridges-2.

Second, competition constraints limited scaling on Bridges-2 to a maximum of four nodes, meaning no more than 8 cores (6.25%) were ever reserved for communication. This minimal overhead likely explains the lack of performance degradation between communication modes.

Based on these findings, we recommend that on systems with a higher number of NUMA domains, polling-based communication is generally more effective than using a dedicated communication thread, as it avoids core reservation overhead.

We also compare both systems using their best-performing configurations. On Bridges-2, we selected two MPI ranks per node with 64 OpenMP threads per rank, using polling-based communication and OpenMPI v5.0.7. On ARCHER2, the best performance was achieved using 16 MPI ranks per node with 8 threads per rank, also with polling-based communication via MPICH v3.4a2. Sustained performance was measured in TFLOP/s at each *last sync point* over the 10 simulated seconds of the Türkiye benchmark.

Figure 5.10: Performance comparison based on the *last sync point* for 10 simulated seconds, using the best configurations for Bridges-2 and ARCHER2 when running the Türkiye benchmark in polling communication mode: 2 MPI ranks per node with 64 OpenMP threads per rank on Bridges-2, and 16 MPI ranks per node with 8 OpenMP threads per rank on ARCHER2.



As shown in Figure 5.10, ARCHER2 consistently outperforms Bridges-2 across all node counts, with the performance gap widening at scale. At four nodes, the two systems exhibit variability of 3.66 and 4.76 GFLOP/s, and relative fluctuations of 0.1% and 1.3%, respectively–demonstrating that ARCHER2 not only delivers higher overall throughput but also maintains significantly more stable and consistent performance throughout the simulation.

Chapter 6

Porting and Performance Optimisation on TeamEPCC Cluster

The final system on which SeisSol was ported and benchmarked is the TeamEPCC Cluster–a two-node GPU-based system housed at the Advanced Computing Facility (ACF) in Edinburgh. The cluster was allocated soley for members of TeamEPCC to support development and testing efforts in preparation for ISC SCC25. Equipped with NVIDIA H100 GPUs, the system provides a platform to evaluate SeisSol's recent GPU support, including the porting process and performance characteristics when running the Türkiye benchmark scenario.

6.1 System Specifications

The TeamEPCC cluster is an HPE Cray XD670 0100 system consisting of two compute nodes (`isc-gpu-01` and `isc-gpu-02`), each equipped with two Intel Xeon Platinum 8468 processors (48 cores per socket) operating at 2.10 GHz. Each socket maps to a distinct NUMA region. Every node is provisioned with 1 TB of DDR5 memory running at 4800 MT/s. There are eight NVIDIA H100 GPUs per node, each featuring 80 GB of High Bandwidth Memory (HMB3) running a driver version 535.247.01 and are connected via NVLink v4 (18-lane NV18). Inter-node communication is handled by Mellanox ConnectX-7 InfiniBand adapters (MT2910 family), supporting link speeds of up to 400 Gb/s.

The system features two main storage partitions: a 7 TB volume managed by a Broadcom MegaRAID 9560 controller, mounted as `/home` on the `isc-gpu-01` node; and a 70 GB NFS-shared partition mounted as `/home2` on `isc-gpu-02`, accessible across both nodes.

The cluster runs Rocky Linux 9.3 (Blue Onyx) with Linux kernel version 5.14. The NVIDIA HPC-X v23.9 software stack is the default and includes CUDA v12.2.91, OpenMPI v4.1.5rc2, UCX v1.15.0 and UCC v1.3.0.

6.2 Porting SeisSol to TeamEPCC Cluster

Porting SeisSol onto a GPU-based system differed slightly from the previous two systems primarly surrounding the kernel code generation and how we run the application due to a different hybrid parallelism strategy. For compilation, we utilise the NVIDIA toolchain using `nvcc` as our CUDA compiler and GCC v11.5.0 for host code as shown in Table 6.1.

Table 6.1: GPU configuration compiled and tested for including build toolchain, dependencies and SeisSol v1.3.1 (commit 57f533) on the TeamEPCC cluster.

Components	**Configuration 1**
Build Toolchains	
Compiler	NVCC v12.2 (GCC v11.5.0 for host)
MPI	OpenMPI v4.1.5
Dependencies	
SeisSol	1.3.1 (57f533)
HDF5	1.12.2
NetCDF	4.9.0
yaml-cpp	0.6.3
Lua	5.4.4
EASI	1.0
ParMETIS	4.0.3
Eigen	3.4.0
GemmForge	1.0
ChainForge	1.0
TensorForge	1.0

We followed the standard procedures for compiling most dependencies and so no major issue arose with the exception of EASI where we received a "relocation `R_X86_64_32` against .rodata" error which had not yet encounterd on other systems. The issue stemmed from EASI compiling without position-independent code (PIC), which is required for proper linking in shared libraries. Recompiling EASI with the `-fPIC` flag resolved the problem, and the SeisSol build proceeded without further linker errors.

Similarly to the other systems, we begin by configuring the CMake build options by selecting the `Release` build mode and specifying the host architecture as `skx`, ensuring

compatibility with Sapphire Rapids and enabling the use of the AVX-512 instruction set. For GPU support, we target the NVIDIA H100 by selecting the `cuda` backend as the device API.

Prior to commit `e6864e`, SeisSol supported GPU offloading via both CUDA and SYCL, the latter requiring additional dependencies including AdaptiveCpp (a SYCL implementation), Boost, and LLVM to be built manually. However, compiling these libraries proved challenging due the vast number of dependencies, leading to the decision to switch to a more recent commit, `57f533`, which removes the SYCL dependency and instead, allows a CUDA-only build. Given that the NVIDIA H100 supports compute capability 9.0, the device architecture was appropriately set to `sm_90`.

CUDA kernel generation in SeisSol relies on the Python-based GemmForge and ChainForge libraries. As of version v1.3.1, this system is undergoing a major restructuring, with both tools being consolidated into a new, still-unreleased library named TensorForge. Until this transition is complete, the existing generators remain in use. The following generators must first be installed `gemmforge`, `chainforge`, and `tensorforge`–must first be installed.

To assess the behaviour of the different kernel generators, we initially attempted to build SeisSol using only GemmForge, following the documentation's claim that ChainForge was optional. To test this, ChainForge was explicitly uninstalled. However, the build process failed due to Python reporting that the ChainForge Python module was missing, revealing that SeisSol still attempted to load ChainForge regardless. This contradicted the documentation and led to further confusion, particularly after developers confirmed in issue #1260 [61] that ChainForge must be removed for exclusive GemmForge usage–yet the build system still expected its presence. Compounding this, another error arose when the auto-generated `gpulike_subroutine.cpp` attempted to include `gemmforge_aux.h`, which was not found, preventing successful compilation.

These failures were ultimately traced back to the omission of two key CMake options: `-DGEMM_TOOLS_LIST` and `-DDEVICE_CODEGEN` which we had not initially specified as this was not documented. When not explicitly set, CMake defaults to the `gemmforge-chainforge` backend generator, even when only `gemmforge` is available, triggering the erroneous dependency on `chainforge`. Additionally, the build flag `-DDEVICE_CODEGEN` accepts only three predefined values: `gemmforge-chainforge`, `tinytc` (for Intel GPUs), and `tensorforge` (a forthcoming unified generator). Notably, no option exists to select `gemmforge` alone, and using the default setting without `chainforge` installed consistently resulted in the aforementioned build errors.

Despite the lack of accurate and complete user documentation, we were able to successfully compile SeisSol using only `gemmforge` by bypassing the documented CMake options. Instead, we directly supplied the `gemmforge` option which is passed to the un-

derlying YaTaeTo code generator library, thereby avoiding the previously encountered errors. In contrast, compiling with `chainforge` is more straightforward, provided that both Python packages are installed and the CMake flags are set appropriately. We additionally enabled `USE_GRAPH_CAPTURING` CMake option to utilise CUDA graphs which where implemented in conjuction with `chainforge` as an optimisation.

6.2.1 Running SeisSol on the Cluster

Running SeisSol on the TeamEPCC cluster differs in that SLURM is not present and thus, additional steps were required with the launch command.

Disabling hyperthreading By default, both nodes have hyperthreading enabled, as confirmed by `lscpu`. We disable it to avoid potential performance degradation from executing on logical rather than physical cores, and to maintain consistency with the two previously tested systems, where SLURM automatically managed CPU topology without using hyperthreading. We accomplished this by executing `sudo echo off > /sys/devices/system/cpu/smt/control` whilst being logged in on both nodes.

MPI Process to GPU Device Binding In contrast to the CPU-based systems, where extensive benchmarking was needed to determine optimal NUMA-aware MPI and OpenMP configurations, the GPU version of SeisSol employs a straightforward one-MPI-rank-per-GPU-die approach. For instance, running on 8 GPUs requires 8 MPI ranks.

To guarantee a correct one-to-one mapping between each MPI rank and GPU, we leverage the OpenMPI-provided `OMPI_COMM_WORLD_LOCAL_RANK` environment variable, which identifies a rank's local index on a given node. This value is then assigned to the `CUDA_VISIBLE_DEVICES` variable, ensuring that each MPI rank is exclusively associated with a single GPU.

Launching a job that fully utilises both nodes (16x NVIDIA H100 GPUs) required manually specifying GPU affinity such that each node only allocates up to eight MPI ranks via the `--host isc-gpu-01:8,isc-gpu-02:8` flag. This ensures both nodes use a balanced number of ranks and that each rank is correctly assigned to a unique device.

OpenMP Thread Allocation Although the core simulation runs on the GPU, small portions of the code–such as the wiggle factor calculation–are executed on the CPU. To accelerate these host-side computations, we evenly distribute the available CPU cores across all MPI ranks. This is done by dividing the total number of cores (96 per node) by the number of GPUs in use. For example, running with

a single MPI rank (using one GPU) assigns all 96 cores to that rank via OpenMP threads. In contrast, when utilising all 8 GPUs on a node (with 8 MPI ranks), each rank is allocated 12 OpenMP threads.

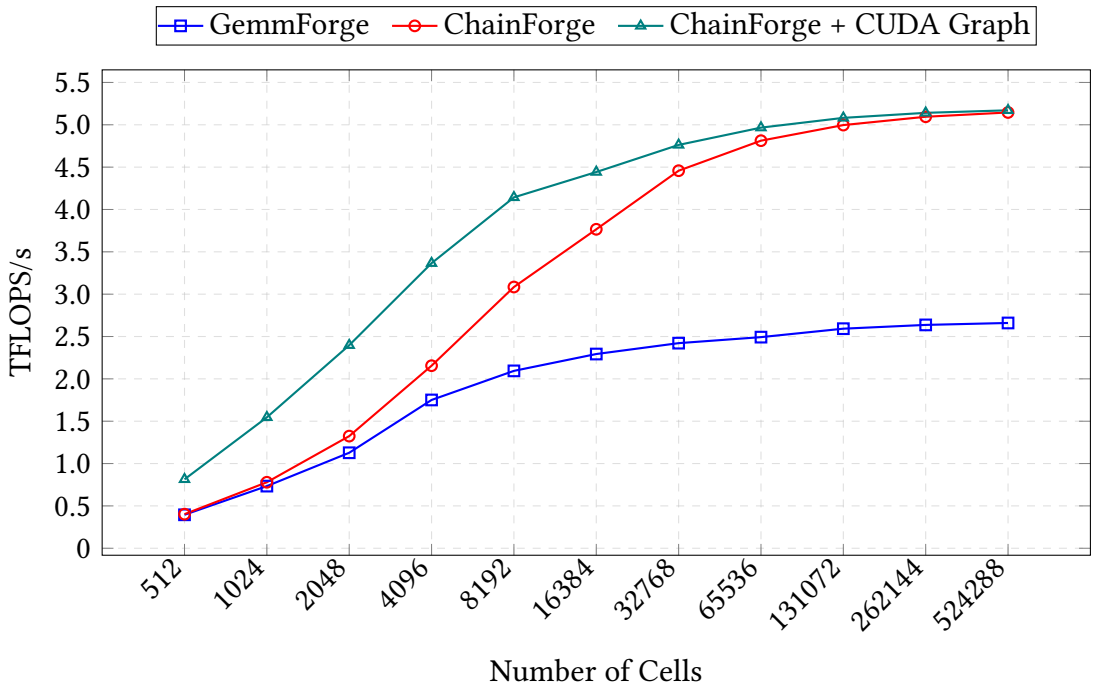
Disable SSH Key Checking Running on more than 8 GPUs would utilise both nodes of the system. However, we encountered a "remote host identification has changed" warning, which prevented us from accessing the other node and thus the run would fail. A workaround for this authentication involves disabling strict host key checking: `--mca plm_rsh_args -o StrictHostKeyChecking=no` which is passed as a command flag to `mpirun`.

File Accessibility Across Nodes By default, the `/home` partition is only accessible from the primary node (`isc-gpu-01`). To prevent "file not found" errors during multi-node runs, we copy the required input test case (e.g., `turkey-benchmark`) and the SeisSol executable to the `/home2` partition, which is shared and accessible by both nodes.

6.2.2 Evaluating Proxy for Kernel Generator Selection

As our first step, we utilised the proxy application, this time to compare across the different available CUDA code generators, `gemmforge` and the newer `chainforge`. We also examine the performance impact of enabling CUDA Graphs.

Figure 6.1: Idealised single-node performance for CUDA code generated kernels using the SeisSol proxy (excluding MPI communication and I/O), compiled with NVCC v12.2 on the TeamEPCC cluster. The benchmark was run with elements between 512 and 524288 and 50 time steps, testing all kernel variants.

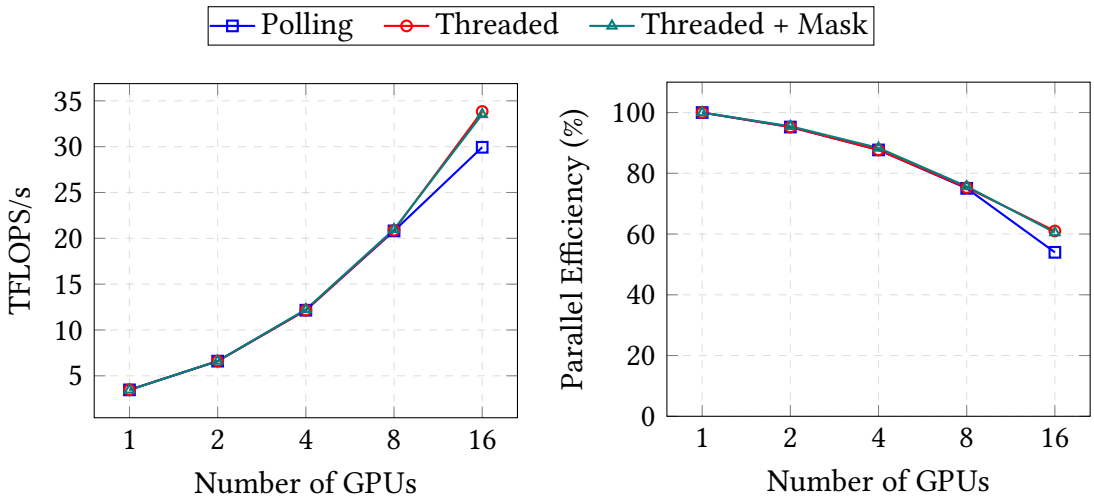


As shown in Figure 6.1, at the largest problem size of 524,288 cells, the original CUDA code generator, `gemmforge`, achieved a peak performance of 2.659 TFLOP/s. In contrast, the newly developed `chainforge` library delivered significantly higher performance, reaching 5.145 TFLOP/s–a 93.5% improvement. This gain is primarily attributed to optimisations from prior work that reduced kernel launch overhead by fusing multiple kernels into a single, larger kernel. Additionally, enabling CUDA Graphs further improves performance, particularly at smaller problem sizes (512 to 16,384 cells), where we observe an average speedup of 65%. We therefore, adopt the `chainforge` kernel generator with CUDA Graphs enabled as the preferred configuration for all subsequent benchmarks.

6.3 Türkiye Simulation Strong Scaling Analysis

In our standard strong scaling benchmark, we compared polling and threaded communication modes, as well as the effect of explicitly assigning free cores for each MPI rank to handle communication. Within a single node, both modes achieved nearly identical performance, peaking at 20.8 TFLOP/s with deviations of less than 1%. However, when scaling to two full nodes (16 GPUs in total, 8 per node), enabling a dedicated communication thread delivered a clear advantage–reaching 33.8 TFLOP/s, 3.7 TFLOP/s higher than polling mode (see Figure 6.2). This improvement is also reflected in parallel efficiency which was 61% for threaded communication compared to 53% for polling once inter-node communication occurs.

Figure 6.2: Strong scaling (left) and parallel efficiency (right) from 1 to 16 GPUs across two nodes for the Türkiye scenario comparing polling, threaded and threaded with affinity masking communication modes on the TeamEPCC cluster.



6.4 Mitigating GPU Performance Variability

During full-system benchmarking on two interconnected nodes (16x NVIDIA H100 GPUs), we observed substantial run-to-run performance variability despite identical configurations. Using the HPC-X v23.9 stack, repeated Türkiye benchmark runs ranged from 18.9 TFLOP/s to 33.7 TFLOP/s–a gap of nearly 15 TFLOP/s.

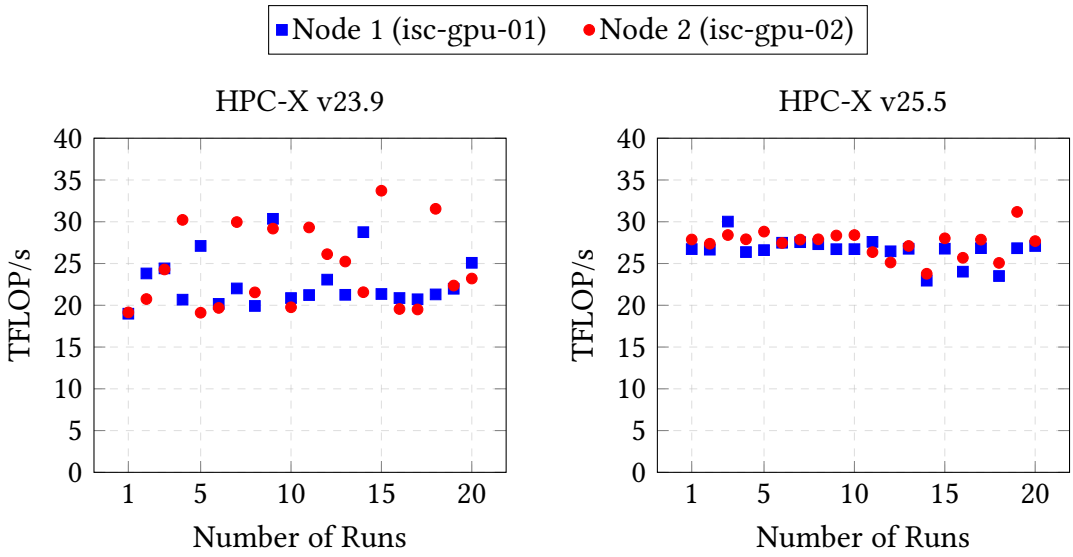
Lower-performing runs (20 TFLOP/s) consistently showed reduced GPU utilisation (55%) and power draw (250–350 W), while higher-performing runs (30 TFLOP/s) reached 85% utilisation and 420 W during the simulation. The NVIDIA Nsight Systems profiler

was used and confirmed these drops corresponded to longer kernel execution times (`flkernelwrapper`, `taylorSumKernel`), indicating reduced hardware saturation.

To test possible causes, we compared GPU-aware MPI data transfer modes: `direct` (GPU–GPU) and `host` (GPU–CPU–GPU staging) via the SeisSol environment variable `SEISSOL_PREFERRED_MPI_DATA_TRANSFER_MODE`. While `direct` achieved higher peaks (30 TFLOP/s), it exhibited larger fluctuations (σ = 5.9 TFLOP/s) than `host` (σ = 2.7 TFLOP/s), which delivered more stable but lower average performance. Full run-by-run results are provided in Appendix C.3.

We also compared two HPC-X releases in `direct` mode: the default v23.9 and the newer v25.5, which includes updated libraries including OpenMPI v4.1.7, UCX v1.18.0, UCC v1.4.3, and HCOLL v4.8.3230. Figure 6.3 shows that v25.5 improved mean performance (26.98 TFLOP/s vs. 23.49 TFLOP/s) and reduced variability (CV = 5.81% vs. 16.56%). This demonstrates that upgrading the runtime stack can yield both higher throughput and more predictable performance.

Figure 6.3: Comparison of HPC-X v23.9 (left) and the more recent HPC-X v25.5 (right) over 20 full two-node runs (16x NVIDIA H100 GPUs) of the Türkiye benchmark showing the performance variability for the TeamEPCC cluster.



6.5 Summary and Outcomes

In summary, our work on the TeamEPCC cluster primarily focused on porting SeisSol to NVIDIA H100 GPUs, benchmarking its performance against prior studies, and evaluating its scalability up to two nodes (up to 16 GPUs). Single-GPU performance measured using the proxy benchmark, was found to be consistent with previous work on the NVIDIA A100 [62, 63]. Specifically, the H100 achieved a peak of 5.15 TFLOP/s, representing an estimated improvement of approximately 1.44 TFLOP/s over the A100, which reached around 3.7 TFLOP/s based on visual approximation from published figures.

Furthermore, with computation and data now residing primarily on the GPU, the NUMA effects observed on ARCHER2's Zen 2 architecture are no longer present. Consequently, on two nodes, the communication thread mode outperformed polling, achieving a peak performance of 33.8 TFLOP/s with the threaded no-mask configuration.

Moreover, the performance variability observed with the system-default HPC-X v23.9 was mitigated by upgrading to HPC-X v25.5. This newer version enabled SeisSol to deliver significantly more stable and consistent results on the TeamEPCC cluster.

6.5.1 Comparison of ARCHER2 and TeamEPCC Systems

We next compare the CPU and GPU performance of ARCHER2 and the TeamEPCC cluster using the proxy application.

Figure 6.4: Performance comparison of SeisSol's wave propagation kernels, computed via the proxy application, on a single ARCHER2 node consisting of two AMD EPYC CPUs and a single NVIDIA H100 GPU on the TeamEPCC cluster.

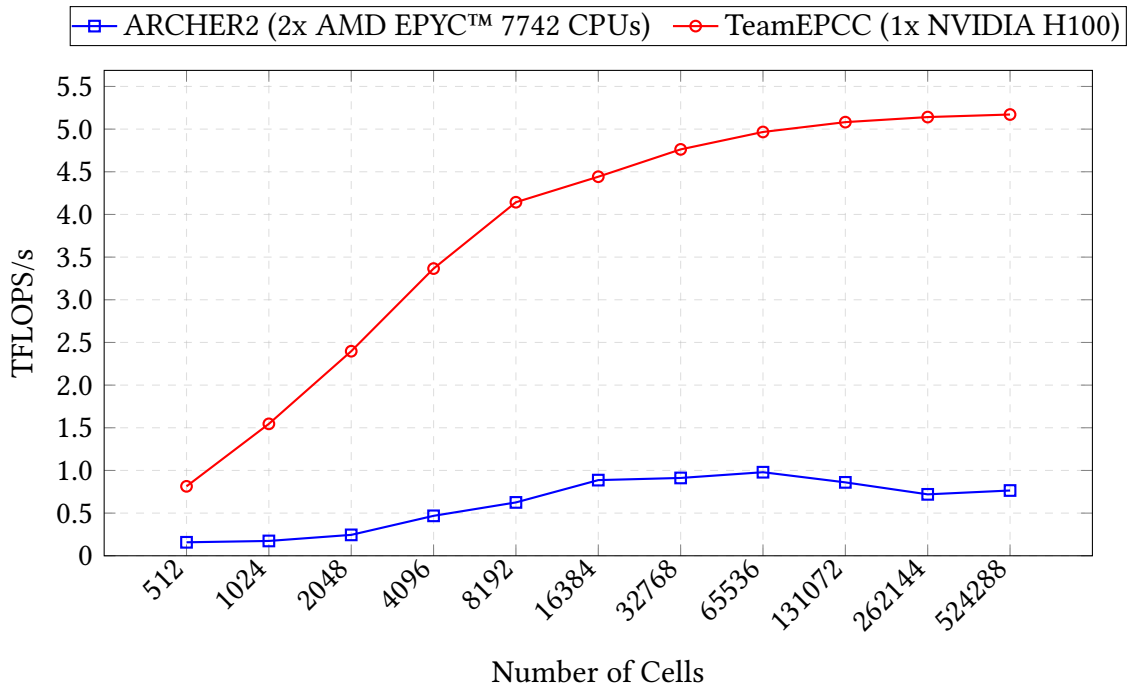


Figure 6.4 highlights the significant performance difference between CPU- and GPU-based systems when running the SeisSol proxy application. On average, the NVIDIA H100 GPU delivers 6.71x higher hardware TFLOP/s and a 4.57x reduction in time-to-solution compared to two AMD EPYC 7742 CPUs. The performance gap widens with increasing problem size, reflecting superior scalability and kernel efficiency on the GPU. In contrast, CPU performance declines beyond 65,536 cells, likely due to memory bandwidth saturation or thread contention. These results suggest that even larger problem sizes could further amplify the GPU's advantage, making this a promising direction for future investigation.

Chapter 7

Conclusions

The dissertation set out to investigate how SeisSol–an earthquake simulator–could be effectively ported, benchmarked, and tuned across three HPC systems: CPU-based Bridges-2 and ARCHER2, as well as the GPU-accelerated TeamEPCC cluster. The project's goals, as defined in the feasibility report, were structured across three objectives. The *Minimum Viable Product* involved participating in the ISC SCC25 competition by tuning SeisSol on the CPU-based Bridges-2 system and subsequently conducting a performance analysis and tuning on ARCHER2. A *Successful Outcome* included extending this investigation to the TeamEPCC cluster using NVIDIA H100 GPUs. Lastly, *Project Extensions* involved exploring emerging and cross-vendor architectures such as NVIDIA's H200 or AMD's MI250.

Throughout this project, we successfully competed in the ISC SCC25 on Bridges-2, where we tuned SeisSol to maximise performance, enabling TeamEPCC to deliver strong competition results. On ARCHER2, systematic testing of hybrid MPI/OpenMP configurations and strong scaling benchmarks revealed that polling communication mode provided the best performance, a behaviour linked to the pronounced NUMA effects of the Zen 2 architecture. Scalability analysis showed that 16 nodes achieved the highest parallel efficiency, while 64 nodes delivered the fastest time-to-solution. We reduced initialisation time by a factor of 5.44 by identifying bottlenecks in ARCHER2's default communication backend and mitigating them through alternative backends. On the TeamEPCC cluster, we successfully ported SeisSol and reproduced previous NVIDIA A100 proxy benchmark results on the newer NVIDIA H100, achieving a 1.39x performance increase. Additionally, we reduced peak performance variability by 10.75% using HPC-X by upgrading its communication library dependencies to more recent versions.

Collectively, these results demonstrate how targeted performance analysis and system-specific tuning can deliver substantial gains and lay a strong foundation for future

optimisation of large-scale scientific simulations. Our results demonstrate a range of tuning techniques that collectively improved SeisSol's performance across the three HPC systems evaluated. In particular, the findings highlight the importance of adapting configurations to the underlying NUMA architecture, as optimal settings on one system may not transfer effectively to another. By systematically benchmarking and tuning SeisSol on Bridges-2, ARCHER2, and the TeamEPCC cluster, we contribute to a broader understanding of its performance characteristics across heterogeneous platforms. These results not only validate SeisSol's scalability on modern HPC infrastructure but also provide actionable guidance for domain scientists and application users seeking to deploy it efficiently. Ultimately, the tuning strategies presented can help accelerate large-scale seismic simulations and support more productive research in computational seismology.

7.1 Reflection

Reflecting on the project, a number of key insights emerged that will help guide and strengthen our future work. One important takeaway is the value of establishing a clear and consistent benchmarking methodology early in the project. For example, while initial single-node evaluations on Bridges-2 used core-scaling for a fixed 10-million element problem, later testing on ARCHER2 and the TeamEPCC cluster adopted a more standardised approach–holding the number of cores constant and varying the problem size between 512 and 524,288 elements. This latter method aligned more closely with established practices in the literature and provided a clearer view of idealised single-node performance. Applying this approach from the outset across all systems would enable consistent and more direct comparisons to be made.

Our NUMA-aware configuration tests also highlighted the importance of validating initial tuning decisions with representative workloads. Early testing on ARCHER2 with the smaller TPV33 benchmark suggested an 8/16 MPI/OpenMP configuration as optimal; however, subsequent evaluations with the more computationally demanding Türkiye benchmark showed that 16/8 delivered better performance. In future projects, conducting these evaluations with both lightweight and production-scale cases from the start will ensure tuning choices remain valid across the full range of problem sizes.

This project also reinforced the importance of realistic resource planning. In our feasibility report, we projected 200 core hours for ARCHER2 which were based on TPV33 runs. However, the Türkiye benchmark ultimately required over 1,000 core hours due to its significantly greater computational cost.

7.2 Future Work

The work presented here provides a strong foundation and opens up numerous potential directions for future research. Given more time, the following areas could be explored:

Revaluating NUMA effects on Bridges-2 During the competition, our tuning investigations indicated that using 2 MPI ranks per node with 64 OpenMP threads delivered the best single-node performance for the TPV33 benchmark. Subsequent benchmarking of the larger Türkiye case was conducted using this configuration. As future work, we aim to revisit our NUMA evaluation by adopting the multi-node hybrid configuration testing methodology used on ARCHER2. Running the larger Türkiye benchmark across different MPI/OpenMP combinations may reveal configurations that outperform our initial choice.

Investigating newer OFI backend on ARCHER2 The default OFI v1.12 (released on March 8th 2021) communication backend on ARCHER2 caused significant initialisation overheads, particularly due to expensive `MPI_Alltoallv` operations. We would like to investigate whether current versions of OFI such as v2.2.0 address these bottlenecks, potentially elimiating SeisSol's initialisation bottlenecks without requiring a switch to UCX.

Identifying cause for performance variability in HPC-X v23.9 While upgrading to the newer HPC-X v25.5 significantly reduced the performance variability observed in SeisSol's GPU execution, further investigation into the root cause of the variability in the system-default HPC-X v23.9 stack would be valuable. Such analysis could reveal bottlenecks within the communication libraries or inefficiencies in SeisSol's GPU implementation that might be optimised.

Optimising asynchronous I/O buffer alignment While the project followed the SeisSol user guide in setting the `async` and `xdmf` buffer alignments to 8 MB, both Bridges-2 and ARCHER2 use a default Lustre stripe size of 1 MB. We would like to investigate whether aligning buffer sizes to the underlying stripe size may potentially further reduce I/O overheads.

Cross-Vendor GPU Performance Analysis To fulfil the original project extension of assessing SeisSol's cross-vendor GPU performance, we could also benchmark additional GPU hardware through the EIDF (NVIDIA H200's) and ARCHER2 (featuring AMD MI250's). This would enable a comparative analysis between vendors, providing deeper insights into SeisSol's portability and performance across GPU-based systems.

Porting SeisSol to the new and upcoming UK HPC System The UK government's

recent announcement of a £750 million investment in a new exascale-capable supercomputer at the University of Edinburgh [64] presents an ideal opportunity to extend this work on a new system. Not only would this help further validate SeisSol's performance portability and scalability, but also contribute to evaluating the system's readiness for exascale workloads.

Bibliography

[1] G. Hager, *Introduction to high performance computing for scientists and engineers*. Crc Press, 2010.

[2] L. A. Steffenel, "Hpc challenges for the next years: the rising of heterogeneity and its impact on simulations," in *Microscopic simulations: forecasting the next two decades*, 2019.

[3] B. Gravelle, "Understanding the performance of hpc applications," 2019.

[4] S. Atchley, C. Zimmer, and Lange, "Frontier: Exploring exascale," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, (New York, NY, USA), Association for Computing Machinery, 2023.

[5] TOP500.org, "The top500 list." https://www.top500.org, 2025. Accessed on: August 7, 2025.

[6] A. Khan, H. Sim, S. S. Vazhkudai, A. R. Butt, and Y. Kim, "An analysis of system balance and architectural trends based on top500 supercomputers," in *The International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia '21, (New York, NY, USA), p. 11–22, Association for Computing Machinery, 2021.

[7] J. Shalf, "Moore's law: the journey ahead," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 378, p. 20190061, Feb. 2020.

[8] S. Rumley, K. Bergman, M. A. Seyedi, and M. Fiorentino, "Evolving requirements and trends of hpc," in *Springer Handbook of Optical Networks*, pp. 725–755, Springer, 2020.

[9] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *IEEE Micro*, vol. 32, no. 3, pp. 122–134, 2012.

[10] P. Navaux, A. Lorenzon, and M. Serpa, "Challenges in high-performance computing," *Journal of the Brazilian Computer Society*, vol. 29, pp. 51–62, 08 2023.

[11] C. Imes, S. Hofmeyr, D. I. D. Kang, and J. P. Walters, "A case study and characterization of a many-socket, multi-tier numa hpc platform," in *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, pp. 74–84, 2020.

[12] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[13] C. Quammen, "Introduction to programming shared-memory and distributed-memory parallel computers," *XRDS: Crossroads, The ACM Magazine for Students*, vol. 8, no. 3, pp. 16–22, 2002.

[14] J. J. Dongarra, S. W. Otto, M. Snir, D. Walker, *et al.*, "An introduction to the mpi standard," *Communications of the ACM*, vol. 18, no. 11, 1995.

[15] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pp. 427–436, 2009.

[16] M. J. Chorley and D. W. Walker, "Performance analysis of a hybrid mpi/openmp application on multi-core clusters," *Journal of Computational Science*, vol. 1, no. 3, pp. 168–174, 2010.

[17] H. Jin, D. Jespersen, P. Mehrotra, R. Biswas, L. Huang, and B. Chapman, "High performance computing using mpi and openmp on multi-core parallel systems," *Parallel Computing*, vol. 37, no. 9, pp. 562–575, 2011. Emerging Programming Paradigms for Large-Scale Scientific Computing.

[18] Linaro Ltd., *Linaro MAP: Parallel Performance Profiler*, 2025. Version 2025.0. Available at: https://www.linaroforge.com/linaro-map.

[19] Hewlett Packard Enterprise, *Cray Performance Measurement and Analysis Tools User Guide*, 2023. Accessed: 2025-08-15.

[20] NVIDIA Corporation, *NVIDIA Nsight Systems User Guide*, 2025. Accessed: 2025-08-15.

[21] NVIDIA Corporation, *NVIDIA Nsight Compute User Guide*, 2025. Accessed: 2025-08-15.

[22] A.-A. Gabriel, V. Kurapati, Z. Niu, N. Schliwa, D. Schneller, T. Ulrich, R. Dorozhinskii, L. Krenz, C. Uphoff, S. Wolf, A. Breuer, A. Heinecke, C. Pelties, S. Rettenberger, S. Wollherr, and M. Bader, "SeisSol."

[23] Scientific Computing World, "2014 Gordon Bell Prize finalists announced." https://www.scientific-computing.com/news/2014-gordon-bell-prize-finalists-announced, 2014. Accessed: 2024-08-05.

[24] T. C. project, "The eu center of excellence for exascale in solid earth (cheese): Implementation, results, and roadmap for the second phase," *Future Generation Computer Systems*, vol. 146, pp. 47–61, 2023.

[25] S. A. Wirp, A.-A. Gabriel, T. Ulrich, and S. Lorito, "Dynamic rupture modeling of large earthquake scenarios at the hellenic arc toward physics-based seismic and tsunami hazard assessment," *Journal of Geophysical Research: Solid Earth*, vol. 129, no. 11, p. e2024JB029320, 2024. e2024JB029320 2024JB029320.

[26] N. Schliwa, A.-A. Gabriel, and Y. Ben-Zion, "Shallow fault zone structure affects rupture dynamics and ground motions of the 2019 ridgecrest sequence to regional distances," *Journal of Geophysical Research: Solid Earth*, vol. 130, no. 6, p. e2025JB031194, 2025. e2025JB031194 2025JB031194.

[27] K. H. Palgunadi, A. Gabriel, D. I. Garagash, T. Ulrich, N. Schliwa, and P. M. Mai, "Ground-Motion Characteristics of Cascading Earthquakes in a Multiscale Fracture Network," *Bulletin of the Seismological Society of America*, 2025.

[28] M. Dumbser and M. Käser, "An arbitrary high-order discontinuous galerkin method for elastic waves on unstructured meshes — ii. the three-dimensional isotropic case," *Geophysical Journal International*, vol. 167, pp. 319–336, 10 2006.

[29] A. Breuer, A. Heinecke, and M. Bader, "Petascale local time stepping for the ader-dg finite element method," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 854–863, 2016.

[30] C. Uphoff and M. Bader, "Yet Another Tensor Toolbox for Discontinuous Galerkin Methods and Other Applications," vol. 46, no. 4, pp. 1–40.

[31] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "Libxsmm: Accelerating small matrix multiplications by runtime code generation," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 981–991, 2016.

[32] PSpaMM Developers, "Pspamm: Parallel sparse matrix multiplication library." https://github.com/TUM-I5/pspamm, 2023. Accessed: 2025-08-05.

[33] G. Guennebaud, B. Jacob, *et al.*, *Eigen: C++ Template Library for Linear Algebra*. Eigen Project, 2021. Version 3.4.0.

[34] C. Uphoff, S. Rettenberger, M. Bader, E. H. Madden, T. Ulrich, S. Wollherr, and A.-A. Gabriel, "Extreme scale multi-physics simulations of the tsunamigenic 2004 sumatra megathrust earthquake," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, (New York, NY, USA), Association for Computing Machinery, 2017.

[35] R. Dorozhinskii and M. Bader, "Seissol on distributed multi-gpu systems: Cuda code generation for the modal discontinuous galerkin method," in *The International Conference on High Performance Computing in Asia-Pacific Region*, HPCAsia '21, (New York, NY, USA), p. 69–82, Association for Computing Machinery, 2021.

[36] A. Breuer, A. Heinecke, S. Rettenberger, A.-A. Gabriel, and C. Pelties, "Sustained petascale performance of seismic simulations with seissol on supermuc," pp. 1–18, 06 2014.

[37] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing - to thread or not to thread?," in *2008 IEEE International Conference on Cluster Computing*, pp. 213–222, 2008.

[38] S. Team, "Precomputed seissol." https://github.com/SeisSol/precomputed-seissol, 2025. Accessed: 2025-03-07.

[39] Southern California Earthquake Center, "Southern california earthquake center." https://www.scec.org/, 2025. Accessed: 2025-08-12.

[40] "TPV33: The problem, version 33 — a benchmark for dynamic rupture simulations." Southern California Earthquake Center (SCEC) 3D Benchmark Descriptions, 2015. Describes a 3D spontaneous rupture on a vertical strike-slip fault with a low-velocity fault zone in a linear elastic medium.

[41] B. Li, K. H. Palgunadi, B. Wu, C. Suhendi, Y. Zhou, A. Ghosh, and P. M. Mai, "Rupture dynamics and velocity structure effects on ground motion during the 2023 türkiye earthquake doublet," *Communications Earth & Environment*, vol. 6, no. 1, p. 228, 2025.

[42] Z. Jia, Z. Jin, M. Marchandon, T. Ulrich, A.-A. Gabriel, W. Fan, P. Shearer, X. Zou, J. Rekoske, F. Bulut, A. Garagon, and Y. Fialko, "The complex dynamics of the 2023 Kahramanmaraş, Turkey, Mw 7.8-7.7 earthquake doublet," vol. 381, no. 6661, pp. 985–990.

[43] T. ULRICH, A.-A. Gabriel, and M. Marchandon, "Mesh and large input files of SeisSol models of the Kahramanmaraş earthquake doublet published in Jia et al.

(2023) and Gabriel et al. (2023)."

[44] H. A. Council, "Getting started with seissol for isc25 scc." https://hpcadvisorycouncil.atlassian.net/wiki/spaces/HPCWORKS/pages/3278569473/Getting+Started+with+SeisSol+for+ISC25+SCC#Test-Run, 2025. Accessed: 2025-03-07.

[45] International Supercomputing Conference (ISC), "ISC High Performance Conference." https://isc-hpc.com/, 2024. Accessed: 2025-03-08.

[46] P. S. Center, "Bridges-2 | psc." https://www.psc.edu/resources/bridges-2/, 2025. Accessed: 2025-08-08.

[47] T. U. o. E. EPCC, "Epcc: Delivering uk supercomputing and data science excellence to the world." https://www.epcc.ed.ac.uk, 2025. Accessed: 2025-08-08.

[48] J. Ahrens, B. Geveci, and C. Law, "Paraview: An end-user tool for large data visualization," in *Visualization Handbook* (C. D. Hansen and C. R. Johnson, eds.), pp. 717–731, Elsevier, 2005.

[49] S. T. Brown, P. Buitrago, E. Hanna, S. Sanielevici, R. Scibek, and N. A. Nystrom, "Bridges-2: A platform for rapidly-evolving and data intensive research," in *Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions*, PEARC '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[50] HPC Advisory Council, "Getting started with seissol for isc25 scc." https://hpcadvisorycouncil.atlassian.net/wiki/spaces/HPCWORKS/pages/3278569473/Getting+Started+with+SeisSol+for+ISC25+SCC, 2025. Accessed: 2025-05-23.

[51] Unidata, "netcdf-c issue #2473: Allow explicit disable of optional dependencies." https://github.com/Unidata/netcdf-c/issues/2473, 2022. Accessed: 2025-07-16.

[52] CBS Pittsburgh, "Thousands still without power after storms damage homes, knock down trees across region," 2025. Accessed: 2025-05-23.

[53] G. Beckett, J. Beech-Brandt, K. Leach, Z. Payne, A. Simpson, L. Smith, A. Turner, and A. Whiting, "ARCHER2 Service Description," tech. rep., Zenodo, Dec. 2024.

[54] A. Jackson, A. Simpson, and A. Turner, "Emissions and energy efficiency on large-scale high performance computing facilities: Archer2 uk national supercomputing service case study," in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, (New York, NY, USA), p. 1866–1870, Association for Computing Machinery, 2023.

[55] K. Zissimos, V. Papadopoulos, V. Spiliopoulos, S. Tsampas, V. Maglogiannis, G. S. Markomanolis, S. Markidis, and I. Sourdis, "D1.2 Report on intra-node and multi-node optimizations for HPC codes," deliverable, European Commission, Sept. 2020. Contract Number: 828947. Project funded by the European Union's Horizon 2020 Programme. Available at: https://ec.europa.eu/research/participants/documents/downloadPublic?documentIds=080166e5d47210ce&appId=PPGMS.

[56] AMD, *High Performance Computing (HPC) Tuning Guide for AMD EPYC™ 7002 Series Processors*, 2020. Document No. 56827.

[57] H. Judge, "Effective use of mpi+openmp on a cray ex supercomputer," 2022. Conference paper, Cray User Group (CUG) 2022.

[58] J. Irudayasamy, J. F. R. Herrera, E. Belikov, and M. Bareford, "Scalability and Performance of OFI and UCX on ARCHER2,"

[59] P. J. Elahi and C. Meyer, "Full-stack approach to hpc testing," tech. rep., Pawsey Supercomputing Research Centre, Kensington, WA, Australia, 2024. Presented as part of internal testing efforts.

[60] ARCHER2 User Support, "Archer2: Tuning performance." https://docs.archer2.ac.uk/user-guide/tuning/, 2025. Accessed: 2025-08-10.

[61] SeisSol Developers, "Issue #1260: Compilation error of seissol master with spack (gemmforge_aux) ." https://github.com/SeisSol/SeisSol/issues/1260, 2025. Accessed: 2025-07-16.

[62] R. Dorozhinskii, L. Krenz, S. Wolf, G. Hillers, A.-A. Gabriel, P. Lanucara, and M. Bader, "Using code-transformation and CUDA graphs to optimize 3D elastic-acoustic simulations on Nvidia DGX A100,"

[63] R. Dorozhinskii, G. B. Gadeschi, and M. Bader, "Fused GEMMs towards an efficient GPU implementation of the ADER-DG method in SeisSol," vol. 36, no. 12, p. e8037.

[64] UK Government, "Scotland to host UK's national supercomputer as Chancellor confirms £750 million investment.," July 2025. Accessed: July 2025.

Appendix A

Bridges-2

Table A.1: Idealised single-node performance of PSpaMM-generated kernels using the SeisSol proxy (excluding MPI communication and I/O), compiled with GCC v13.3.1 on Bridges-2. The benchmark was run with one million elements and 50 time steps, testing all kernel variants.

CPU Cores	GFLOP/s	Parallel Efficiency
1	24.83	1.0000
2	44.39	0.8941
4	84.21	0.8481
8	170.12	0.8566
16	334.67	0.8426
32	521.24	0.6561
64	678.21	0.4269
128	675.98	0.2127

Table A.2: Evaluating the single-node NUMA effects of running the TPV33 benchmark in polling communication mode by varying the number MPI ranks and OpenMP threads (Processes/Threads) on Bridges-2.

MPI Ranks	OpenMP Threads	Runtime (s)	GFLOP/s
1	128	24.42	274.90
2	**64**	**12.74**	**527.87**
4	32	176.00	38.23
8	16	197.00	34.41
16	8	299.00	23.09

Table A.3: Strong scaling of the Türkiye benchmark on Bridges-2, configured with 2 MPI ranks and 64 OpenMP threads per rank across 1 to 4 nodes. The results compare performance across communication modes (polling vs. threaded) and OpenMP thread affinity strategies.

	Polling		Threaded	
Nodes	Spread	Close	Spread	Close
1	906.23	852.90	**918.48**	916.60
2	1745.90	1456.53	1726.50	**1782.30**
3	2523.90	**2619.20**	2561.60	2572.00
4	**3226.10**	2916.60	3208.90	3202.30

Table A.4: Strong scaling of the Türkiye benchmark on Bridges-2 using 2 MPI ranks per node and 64 OpenMP threads per rank, scaled from 1 to 4 nodes. Results compare performance across OpenMPI v4.0.5, OpenMPI v5.0.5, and MVAPICH2 v2.3.0.

Nodes	OpenMPI v4.0.5	OpenMPI v5.0.7	MVAPICH v2.3.0
1	**906.23**	892.52	372.69
2	1745.90	**1762.90**	606.36
3	2523.90	**2703.20**	975.60
4	3226.10	**3530.80**	1221.60

Appendix B

ARCHER2

Table B.1: Idealised single-node performance across nine compiler-kernel generator configurations using the SeisSol proxy (excluding MPI communication and I/O) on ARCHER2. The benchmarks were run between 512 and 524288 elements for 100 timesteps, testing all kernels.

		Eigen				PSpaMM				LIBXSMM			
	Size	Time (s)	GFLOPS (HW)	GFLOPS (NZ)	GiB/s	Time (s)	GFLOPS (HW)	GFLOPS (NZ)	GiB/s	Time (s)	GFLOPS (HW)	GFLOPS (NZ)	GiB/s
GCC	512	0.035	209.809	82.906	35.781	0.041	156.921	62.007	26.762	0.041	157.986	62.428	26.943
	1024	0.076	170.777	67.524	29.125	0.075	173.152	68.463	29.530	0.075	173.765	68.705	29.634
	2048	0.109	239.108	94.518	40.778	0.103	253.704	100.288	43.267	0.107	244.945	96.825	41.774
	4096	0.121	430.982	170.439	73.501	0.132	419.769	166.005	71.589	0.112	468.020	185.087	79.818
	8192	0.203	520.080	205.643	88.696	0.164	669.994	264.920	114.263	0.177	625.273	247.237	106.636
	16384	0.358	586.065	231.770	99.949	0.231	910.649	360.133	155.305	0.237	886.615	350.629	151.207
	32768	0.668	631.740	249.848	107.739	0.451	932.630	368.848	159.054	0.464	911.976	360.680	155.532
	65536	1.262	665.514	263.290	113.499	0.848	993.134	392.902	169.373	0.858	979.141	387.367	166.986
	131072	2.760	610.071	241.371	104.044	2.025	829.902	328.347	141.535	1.953	860.202	340.334	146.702
	262144	6.314	532.427	210.644	90.802	4.604	729.690	288.688	124.444	4.668	719.810	284.779	122.759
	524288	12.527	536.561	212.282	91.507	8.946	751.235	297.214	128.118	8.786	764.887	302.615	130.447
AOCC	512	0.020	410.270	162.118	69.969	0.008	742.027	293.212	126.548	0.008	737.102	291.266	125.708
	1024	0.039	443.351	175.297	75.611	0.033	633.672	250.549	108.069	0.014	879.978	347.936	150.075
	2048	0.101	258.788	102.298	44.134	0.098	265.235	104.846	45.234	0.100	262.051	103.587	44.691
	4096	0.142	387.192	153.122	66.033	0.108	485.314	191.926	82.767	0.135	421.636	166.743	71.907
	8192	0.203	516.730	204.318	88.125	0.145	740.130	292.652	126.224	0.153	714.723	282.606	121.891
	16384	0.335	626.551	247.782	106.854	0.235	894.666	353.813	152.580	0.269	786.749	311.135	134.175
	32768	0.646	650.244	257.166	110.895	0.435	965.846	381.985	164.719	0.427	982.373	388.521	167.538
	65536	1.270	661.133	261.557	112.752	0.831	1010.479	399.765	172.331	0.847	991.933	392.427	169.168
	131072	2.752	610.954	241.721	104.194	1.999	840.674	332.608	143.372	1.967	854.282	337.992	145.692
	262144	6.061	554.479	219.369	94.563	4.599	730.513	289.013	124.584	4.624	725.802	287.456	123.913
	524288	11.742	572.332	226.434	97.607	8.509	789.741	312.448	134.685	8.646	777.226	307.497	132.347
CRAY	512	0.016	394.555	155.908	67.289	0.013	499.837	197.510	85.244	0.013	498.125	196.834	84.952
	1024	0.027	478.756	189.296	81.649	0.019	687.746	271.929	117.291	0.019	690.060	272.844	117.685
	2048	0.100	262.034	103.581	44.688	0.102	255.555	101.019	43.583	0.102	255.647	101.056	43.599
	4096	0.113	462.157	182.768	78.818	0.112	468.853	185.416	79.960	0.103	506.071	200.135	86.307
	8192	0.204	517.147	204.483	88.196	0.161	650.438	257.188	110.928	0.174	607.688	240.284	103.637
	16384	0.315	667.716	264.061	113.875	0.225	930.172	367.854	158.635	0.249	846.388	334.720	144.346
	32768	0.598	701.714	277.523	119.673	0.414	1014.679	401.298	173.047	0.407	**1034.093**	408.976	176.358
	65536	1.177	713.759	282.376	121.727	0.840	1004.391	397.356	171.292	0.830	1013.295	400.878	172.811
	131072	2.516	667.761	264.196	113.882	1.911	879.374	347.920	149.972	1.954	859.543	340.074	146.590
	262144	5.569	603.341	238.700	102.896	4.486	749.105	296.369	127.755	4.631	725.453	287.012	123.721
	524288	10.755	624.850	247.212	106.564	8.394	800.646	316.763	136.545	8.692	773.162	305.889	131.858

Table B.2: Standard deviations and coefficient of variation (CV) for single-node performance across nine compiler and kernel generator configurations using the SeisSol proxy (excluding MPI communiation and I/O) on ARCHER2. The benchmarks were run between 512 and 524288 elements for 100 timesteps, testing all kernels.

Compiler	Standard Deviation (GFLOP/s)			CV (%)		
	Eigen	PSpaMM	LIBXSMM	Eigen	PSpaMM	LIBXSMM
CRAY	143.80	240.36	233.67	25.96	33.30	32.91
GCC	178.96	313.95	308.27	38.35	50.63	49.92
AOCC	**128.12**	**214.99**	**226.34**	**24.76**	**29.20**	**31.17**

Table B.3: Strong scaling of the simulation (excluding initialisation) across varying node counts for the Türkiye scenario comparing six MPI/OpenMP configurations under the polling communication mode on ARCHER2.

Nodes	MPI Ranks	OpenMP Threads	Simulation Time (s)	GFLOP/s
4 Nodes				
	2	64	541	3,609.00
	4	32	495	3,943.60
	8	16	476	4,100.10
	16	**8**	**456**	**4,281.00**
	32	4	457	4,278.90
	64	2	574	3,410.10
8 Nodes				
	2	64	286	6,826.90
	4	32	255	7,644.70
	8	16	245	7,970.40
	16	8	229	8,507.30
	32	**4**	**226**	**8,652.50**
	64	2	286	6,858.80
16 Nodes				
	2	64	149	13,086.60
	4	32	133	14,665.80
	8	16	120	16,257.40
	16	8	111	17,499.0
	32	**4**	**110**	**17,822.70**
	64	2	149	13,086.60
32 Nodes				
	2	64	79	24,582.60
	4	32	72	27,149.50
	8	16	61	31,799
	16	**8**	**54**	**35,711.40**
	32	4	55	35,689
	64	2	73	27,126.20

Table B.4: Strong scaling benchmark for the top three performing hybrid configurations from 1 to 128 nodes comparing polling and threaded communication modes for the Türkiye benchmark on ARCHER2.

		Polling			Threaded		
MPI/OpenMP	**Nodes**	**Time (s)**	**GFLOPS (HW)**	**Efficiency**	**Time (s)**	**GFLOPS (HW)**	**Efficiency**
8/16	1	3741.13	1,043.87	1.000	3900.60	1,005.37	1.000
	2	1884.10	2,073.50	0.993	1987.71	1,965.57	0.977
	4	946.37	4,129.70	0.989	1014.23	3,853.63	0.958
	8	485	8,060.00	0.965	528	7,404.50	0.920
	16	241	16,190.70	0.969	282	13,866.50	0.862
	32	119	32,669.40	0.978	145	26,845.00	0.834
	64	57.8	67,876.00	1.015	86	45,655.60	0.709
	128	33.8	116,385.00	0.871	63.8	61,787.20	0.480
16/8	1	3673	1,063.30	1.000	4,118	948.3201	1.000
	2	1829	2,136.20	1.004	2,132	1,832.40	0.966
	4	908	4,305.30	1.012	1,091	3,581.10	0.944
	8	460	8,504.20	0.999	573	6,817	0.898
	16	223	17,497.80	1.028	289	13,555.80	0.893
	32	109	35,770.60	1.051	162	24,147.20	0.795
	64	55	71,291.50	1.047	100	39,228	0.646
	128	31	126,593.40	0.930	73	54,113.40	0.445
32/4	1	3690	1,058.70	1.000	4,100	952.8227	1.000
	2	1920	2,147.40	1.014	2,144	1,823.40	0.956
	4	920	4,252.00	1.004	1,107	3,532.50	0.926
	8	462	8,470.50	1.000	567	6,908.30	0.906
	16	220	17,782.30	1.049	332	11,812	0.774
	32	111	35,393.10	1.044	190	20,670.50	0.677
	64	58	68,067.70	1.004	133	29,785	0.488
	128	33	119,842.00	0.884	264	29,683.60	0.243

Table B.5: Callstack as reported by CrayPat v22.12.0 showing `MPI_Alltoallv` as the most expensive function which is the primary factor resulting in the relatively slow initialisation times.

Function Name	Callstack

`initSeisSol::initMesh()`	`readMeshPUML`
	`  PUMLReader::generatePUML`
	`  PUML::generateMesh`
	`  MPI_Alltoallv`
	`  PUML::generatedSharedAndGID<>`
	`  MPI_Alltoallv`
	`  PUML::partition`
	`  MPI_Alltoallv`

`initSeisSol::initIO()`	`setupOutput`
	`  WaveFieldWriter::init`
	`  WaveFieldWriterExecutor::execInit`
	`  ParallelVertexFilter::filter`
	`  MPI_Alltoallv`
	`initFaultOutputManager`
	`  FaultWriter::init`
	`  FaultWriterExecutor::execInit`
	`  MPI_Alltoallv`

Table B.6: Full runtime parallel efficiency of the Türkiye benchmark using polling communication mode, comparing OFI with startup flags and UCX against the default OFI configuration. All runs use 16 MPI ranks per node and 8 OpenMP threads per rank from 1 to 128 nodes on ARCHER2.

Nodes	OFI	OFI+Flags	UCX
1	1.0000	1.0000	1.0000
2	0.9995	0.9863	0.9917
4	0.9984	0.9943	0.9725
8	0.9691	0.9773	0.9797
16	0.9320	0.9943	1.0008
32	0.8560	0.9529	0.9756
64	0.5556	0.8695	0.8932
128	0.3107	0.5936	0.5279

Table B.7: Performance comparison based on the *last sync point* for 10 simulated seconds, using the best configurations for Bridges-2 and ARCHER2 when running the Türkiye benchmark in polling communication mode: 2 MPI ranks per node with 64 OpenMP threads per rank on Bridges-2, and 16 MPI ranks per node with 8 OpenMP threads per rank on ARCHER2.

	ARCHER2			Bridges-2		
Simulated Second	**1 Node**	**2 Nodes**	**4 Nodes**	**1 Node**	**2 Nodes**	**4 Nodes**
1	1.0543	2.0850	4.1335	0.8920	1.7636	3.6393
2	1.0582	2.0977	4.1461	0.8940	1.7702	3.6561
3	1.0575	2.1112	4.1456	0.8910	1.7662	3.6564
4	1.0571	2.1156	4.1442	0.8930	1.7711	3.6567
5	1.0576	2.1060	4.1452	0.8930	1.7656	3.6521
6	1.0568	2.1053	4.1465	0.8910	1.7652	3.6473
7	1.0592	2.1040	4.1478	0.8910	1.7683	3.6196
8	1.0598	2.1098	4.1451	0.8930	1.7340	3.5826
9	1.0590	2.1052	4.1451	0.8910	1.7522	3.5127
10	1.0583	2.1059	4.1453	0.8910	1.7742	3.5826

Appendix C

TeamEPCC Cluster

Table C.1: Idealised single-node performance for CUDA code generated kernels using the SeisSol proxy (excluding MPI communication and I/O), compiled with NVCC v12.2 on the TeamEPCC cluster. The benchmark was run with elements between 512 and 524288 and 50 time steps, testing all kernel variants.

		GemmForge				ChainForge			
	Size	**Time (s)**	**GFLOPS (HW)**	**GFLOPS (NZ)**	**GiB/s**	**Time (s)**	**GFLOPS (HW)**	**GFLOPS (NZ)**	**GiB/s**
No CUDA Graph	512	0.023	363.994	108.878	52.701	0.023	402.478	138.131	53.0051
	1024	0.024	707.468	211.719	102.432	0.024	779.243	267.537	102.623
	2048	0.028	1201.383	359.777	173.944	0.028	1324.110	454.854	174.381
	4096	0.035	1936.563	579.584	280.388	0.035	2155.343	740.0376	283.851
	8192	0.049	2792.570	835.256	404.327	0.049	3084.493	1058.536	406.218
	16384	0.079	3478.146	1040.360	503.589	0.081	3766.076	1292.490	495.979
	32768	0.135	4087.553	1222.680	591.823	0.136	4457.826	1529.940	587.081
	65536	0.249	4449.623	1331.356	644.246	0.253	4812.323	1651.963	633.767
	131072	0.478	4639.643	1388.266	671.758	0.488	4996.456	1715.230	658.016
	262144	0.935	4750.246	1421.270	687.772	0.958	5094.706	1748.870	670.956
	524288	1.844	4818.483	1441.586	697.652	1.898	5145.396	1766.170	677.632
CUDA Graph	512	0.021	395.051	118.168	57.198	0.011	813.747	279.279	107.168
	1024	0.023	733.702	219.569	106.230	0.012	1545.930	530.763	203.594
	2048	0.030	1127.536	337.663	163.252	0.015	2397.166	823.469	315.699
	4096	0.039	1750.883	524.014	253.504	0.022	3365.636	1155.590	443.243
	8192	0.066	2094.316	626.408	303.229	0.036	4143.056	1421.820	545.627
	16384	0.121	2293.243	685.939	332.031	0.068	4442.046	1524.480	585.002
	32768	0.229	2421.706	724.389	350.630	0.128	4762.243	1634.413	627.171
	65536	0.445	2492.720	745.835	360.912	0.245	4966.403	1704.856	654.059
	131072	0.856	2592.486	775.720	375.357	0.480	5081.670	1744.486	669.239
	262144	1.685	2637.053	789.004	381.810	0.950	5141.046	1764.773	677.058
	524288	3.341	2659.963	795.804	385.127	1.889	5170.936	1774.940	680.995

Table C.2: Strong scaling and parallel efficiency from 1 to 16 GPUs across two nodes for the Türkiye benchmark comparing polling, threaded and threaded with affinity masking communication modes on the TeamEPCC cluster.

	Polling		Threaded		Threaded + Mask	
GPUs	**TFLOP/s**	**Efficiency**	**TFLOP/s**	**Efficiency**	**TFLOP/s**	**Efficiency**
1	3465.9	1.0000	3467.2	1.0000	3463.2	1.0000
2	6600.8	0.9522	6604.4	0.9524	6613.1	0.9548
4	12156.5	0.8769	12160.8	0.8768	12238.4	0.8835
8	20807.7	0.7504	20882.9	0.7529	20970.3	0.7569
16	29933.3	0.5398	33842.2	0.6100	33521.4	0.6050

Table C.3: Performance comparison between `direct` and `host` data transfer modes (in TFLOP/s) across six runs of the Türkiye benchmark on the TeamEPCC cluster with signifiant deviations highlighted

Run #	**Direct (TFLOP/s)**	**Host (TFLOP/s)**
1	29.62	21.62
2	32.61	23.38
3	**20.30**	24.49
4	29.84	22.69
5	**18.95**	**28.12**
6	**20.95**	27.92

Table C.4: Comparison of HPC-X v23.9 (left) and the more recent HPC-X v25.5 (right) over 20 full two-node runs (16x NVIDIA H100 GPUs) of the Türkiye benchmark showing the performance variability for the TeamEPCC cluster.

	HPC-X v23.9		HPC-X v25.5	
Run No.	**Node 1**	**Node 2**	**Node 1**	**Node 2**
1	18.99	19.15	26.73	27.88
2	23.81	20.75	26.66	27.37
3	24.42	24.29	30.02	28.41
4	20.67	30.22	26.37	27.90
5	27.10	19.11	26.60	28.82
6	20.17	19.68	27.49	27.46
7	22.02	29.96	27.55	27.86
8	19.92	21.55	27.31	27.90
9	30.35	29.18	26.72	28.35
10	20.86	19.76	26.72	28.41
11	21.22	29.32	27.59	26.36
12	23.07	26.12	26.46	25.12
13	21.25	25.24	26.77	27.10
14	28.75	21.57	22.95	23.78
15	21.35	33.71	26.78	28.02
16	20.87	19.55	24.02	25.69
17	20.72	19.50	26.84	27.86
18	21.30	31.55	23.50	25.07
19	21.98	22.36	26.83	31.18
20	25.08	23.20	27.10	27.68