Testing and Benchmarking Emerging Supercomputers via the MFC Flow Solver

Benjamin Wilfong
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, GA, USA
bwilfong3@gatech.edu

Anand Radhakrishnan
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, GA, USA
aradhakr34@gatech.edu

Henry Le Berre
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, GA, USA
henryleberre@gatech.edu

Tanush Prathi
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, GA, USA
tprathi3@gatech.edu

Stephen Abbott
Hewlett Packard Enterprise (HPE)
Saint Paul, USA
stephen.abbott@hpe.com

Spencer H. Bryngelson
School of Computational Science and Engineering
Georgia Institute of Technology
Atlanta, GA, USA
shb@gatech.edu

Abstract

Deploying new supercomputers requires testing and evaluation via application codes. Portable, user-friendly tools enable evaluation, and the Multicomponent Flow Code (MFC), a computational fluid dynamics (CFD) code, addresses this need. MFC is adorned with a toolchain that automates input generation, compilation, batch job submission, regression testing, and benchmarking. The toolchain design enables users to evaluate compiler–hardware combinations for correctness and performance with limited software engineering experience. As with other PDE solvers, wall time per spatially discretized grid point serves as a figure of merit. We present MFC benchmarking results for five generations of NVIDIA GPUs, three generations of AMD GPUs, and various CPU architectures, utilizing Intel, Cray, NVIDIA, AMD, and GNU compilers. These tests have revealed compiler bugs and regressions on recent machines such as Frontier and El Capitan. MFC has benchmarked approximately 50 compute devices and 5 flagship supercomputers.

CCS Concepts

• **Hardware → Testing with distributed and parallel systems**; • **Computing methodologies → Massively parallel and high-performance simulations**; • **Networks** → *Network performance evaluation*; • **Applied computing** → Physics.

Keywords

Testing, Benchmarking, Directives, Computational Fluid Dynamics

1 Introduction

Supercomputer tests and benchmarks rely on portable and performant software applications to make meaningful comparisons between new and existing systems and hardware. This work presents MFC [9] as an application that addresses this need. MFC is a GPU-accelerated [25], feature-rich [35], portable [37], and user-friendly computational fluid dynamics (CFD) code used to test and benchmark five generations of NVIDIA GPUs, three generations of AMD GPUs, and many CPUs.

MFC is a multiphysics flow solver that has been used to simulate compressible multi-species, phase, and chemically reacting fluid flows [7, 8, 10, 11, 24]. The spatiotemporal requirements of compressible multiphysics flow simulations have driven the authors to prioritize performance and portability in the design of MFC. The background of researchers in the field has also led us to prioritize user-friendliness and approachability in MFC's design. This work describes how MFC's predictable performance and user-friendly interface make it a reliable and approachable application for testing and benchmarking new supercomputers.

The MFC toolchain is designed to be user-friendly and portable, helping users test and benchmark HPC systems with minimal knowledge of the underlying hardware or software. The user interacts with the toolchain via a wrapper script that requires a one-time setup to add support for an alternative system. The user completes setup by identifying and specifying the required Lmod [20] modules and shell environment variables. The final step is to create a system-specific job-submission template file that supports multiple schedulers and their idiosyncrasies. The bash wrapper automates the process of loading modules, building MFC and its dependencies, running regression tests, and benchmarking the code once the initial setup is complete. The toolchain is designed to be easily adapted and updated by users. This strategy lets users modify test cases and benchmarks to evaluate code and language features, confirm system-specific correctness, and identify performance bottlenecks.
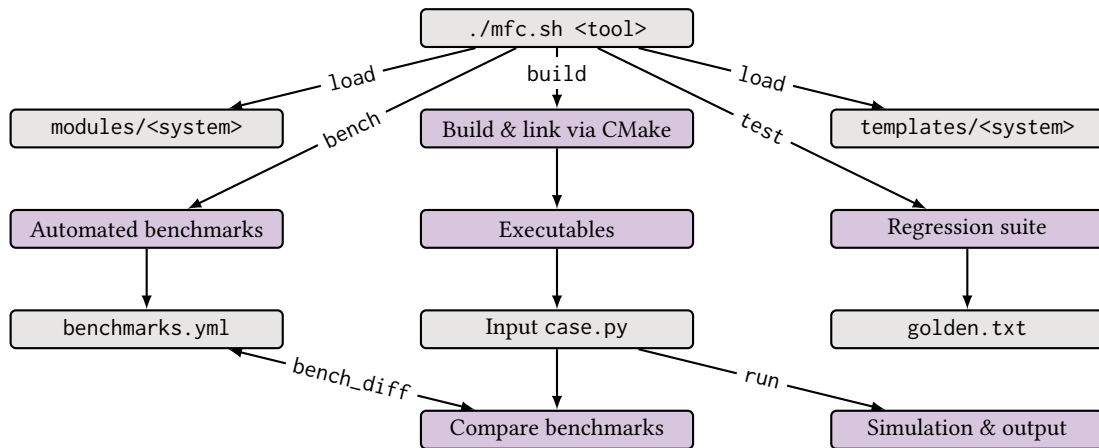
Figure 1: The MFC toolchain and its connectivity.

We summarize MFC's performance with a single figure of merit, *grindtime*: nanoseconds of wall time per grid point, equation, and right-hand-side evaluation. Here, the grid points represent the spatial discretization points of the simulation domain, the equations refer to the system of partial differential equations solved by the code, and the right-hand side evaluations denote the operations performed to advance the solution in time. This definition provides a figure that describes the time it takes to perform the smallest measurable unit of work in an application that solves time-dependent partial differential equations (PDEs). Defining the grindtime in terms of wall time means it follows strong scaling trends when increasing device count. We compare a single GPU driven by one MPI rank with a single CPU die driven by multiple. Expressing performance per smallest measurable unit makes grindtime independent of problem size, the number of physical model equations, and the time-integration scheme.

The grindtime measurement accounts for MPI communication and host–device transfers relevant to network, CPU, and offload device (e.g., GPU) performance. It neglects the time spent performing code initialization and I/O operations. I/O costs are not directly benchmarked in the present work as they are sufficiently small compared to compute costs. Still, MFC writes an I/O profile for each case, which can be used to evaluate I/O performance or bottlenecks if unexpected behavior is observed. Defining grindtime in this way evaluates how hardware and network performance impact the run-time of core compute kernels.

Manuscript structure: MFC's suggested role in the existing landscape of HPC testing and benchmarking tools is described in section 2. Section 3 describes the steps for testing and benchmarking a supercomputer with MFC. Section 4 and section 5 provide details on how users can extend the testing and benchmarking capabilities of MFC and give examples of bugs and performance bottlenecks identified by each tool. Application of MFC as a tool for system deployment and testing via a standardized benchmark case, weak scaling, and strong scaling is described in section 6. Limitations, implications, and final thoughts are given in section 7 and section 8.

2 Existing tools

Many tools support the benchmarking of HPC systems. One such tool is the High-Performance Linpack (HPL) benchmark [14], which solves a dense system of linear equations via LU factorization with partial pivoting. The HPL benchmark estimates a supercomputer's maximum sustained performance, which informs the semiannual TOP500 ranking [14]. HPL is widely accepted but limited in scope, representing only a fraction of user applications. The SPEChpc benchmark suite [17] was created, in part, as a response to this limitation. The SPEChpc benchmarks comprise multiple maintained user applications that vary in subject matter, numerical methods, and programming models, providing a more holistic measure of system performance.

Benchmarking and automation tools can help users test HPC systems. A non-exhaustive list of such tools includes ReFrame [12], JUBE [19], Ramble [16], BenchPRO [29], and OLCF Test Harness [22]. Each tool has strengths, but all aim to provide automated interfaces for testing and benchmarking supercomputers and their hardware. MFC itself is not a suitable replacement for any of these tools. MFC is, however, a suitable application for testing and benchmarking HPC systems with these tools. The user-friendly interface and automated building, testing, and benchmarking processes in MFC make it an easy-to-integrate candidate for use with these existing tools.

3 Tooling summary and usage

The core of MFC's user-friendly interface is the bash wrapper `mfc.sh` that provides quick access to all of MFC's automated tools. The control flow of the toolchain is shown in fig. 1, and table 1 lists the tools for testing and benchmarking a new system. The following sections describe how a user would typically use each of these tools to test and benchmark a new system using MFC.

Step 1: System and environment setup. The first step in testing and benchmarking a new system with MFC is to set up the compute environment. Setting up the environment starts with identifying the relevant modules to load and environment variables to set. The

Table 1: List of relevant automated tools accessible via the wrapper script `mfc.sh`. The tools in this list are in the order a user would typically use them to test and benchmark a new system with MFC. Each tool has additional command-line options accessible via `./mfc.sh <tool> --help`.

Tool	Description
load	Load modules and initialize environment
build	Build MFC's source and dependencies
test	Run the regression test suite
bench	Run the benchmark suite
bench_diff	Compare benchmark results
run	Run a user-defined case file

required modules for currently supported clusters and supercomputers are listed in `toolchain/modules` and are easy to extend. Listing 1 shows an example entry for NCSA Delta, which supports CPU and GPU builds with different modules and environment variables. Line 1 assigns the system name to the identifier d to be used in the command line interface. Modules and environment variables used by both CPU and GPU builds are stored in the `d-all` entry and loaded first. The `d-cpu` and `d-gpu` entries store modules and environment variables specific to building and running MFC on CPU or GPU hardware. Once the relevant modules and environment variables are identified, the user can load them with the command `source ./mfc.sh load`. This command prompts the user for the system identifier (for example (d) for NCSA Delta), and configuration ((c | cpu) for CPU builds and (g | gpu) for GPU builds). Executing `source ./mfc.sh load` is configured to purge loaded modules and load the modules and environment variables appropriate for that system.

Listing 1: Example module and environment variables loaded into the user's environment for NCSA Delta.

```
1  d       NCSA Delta
2  d-all python/3.11.6
3  d-cpu gcc/11.4.0 openmpi
4  d-gpu nvhpc/24.1 cuda/12.3.0 openmpi/4.1.5+cuda
         cmake
5  d-gpu CC=nvc CXX=nvc++ FC=nvfortran
6  d-gpu MFC_CUDA_CC=80,86
```

The final step in setting up the environment is to create a system-specific template file in the `toolchain/templates/` directory. MFC uses the Mako templating library [3] for system-specific templates that create Bash scripts, which can run MFC executables in interactive or batch mode, depending on the user's needs.

System-specific templates are used because they provide a way to support multiple scheduling systems, such as Slurm, PBS, LSF, and Flux, without requiring future users to be familiar with the details of the scheduling system. Templates can also set additional run-time environment variables and settings that are irrelevant to compilation. For example, the template for OLCF Frontier sets the environment variable `MPICH_GPU_SUPPORT_ENABLED=1` to activate GPU-aware MPI at run-time. The template also sets `ulimit`

`-s unlimited` to expand the stack size for simulating particularly large problems. The template files help the toolchain add the commands to perform high-level and kernel-level profiles to interactive and batch scripts. This approach removes the need for direct user interaction with profiling tools. Once the module and template files are created for the new system, one can build MFC and its dependencies.

Step 2: Building. Once the relevant modules are loaded and environment variables set, the user can build MFC and its dependencies with the command `./mfc.sh build <config>`, where `<config>` is `--gpu acc|mp` for GPU builds and `--no-gpu` for CPU builds. GPU acceleration is provided via OpenACC [34], which supports GPU offloading for NVIDIA and AMD GPUs, or OpenMP [23], the latter being better supported by compilers targeting AMD and Intel GPUs. MFC's dependencies vary by system and hardware. All MFC simulations depend on `silo` and `hdf5` for visualization and post-processing. `silo` and `hdf5` are fetched and compiled automatically by CMake for the specific hardware and system configuration. Some features in MFC rely on fast Fourier transforms (FFTs), which are provided by FFTW [15] for CPU builds, cuFFT [21] for NVIDIA GPU builds, and `hipFFT` [1] for AMD GPU builds. CMake automatically detects the hardware and system configuration and builds the appropriate Fourier transform library. Then, the user tests the build.

Step 3: Regression testing: With the system environment set up and the source code built, MFC's test suite can now be run to identify any run-time errors or correctness bugs resulting from the hardware-software configuration. The test suite is run with the command `./mfc.sh test -- -c <system>` where `<system>` is the name of the mako template file created in step 1. At the time of writing, MFC's test suite comprises approximately 500 test cases that cover most available features. Section 4 describes details of the regression suite, including how cases are added, how results are compared, and how the suite has aided in identifying and correcting compiler-hardware bugs. The user can proceed to the benchmarking step once a hardware-compiler combination has been tested for correctness.

Step 4: Benchmarking. MFC contains two benchmarking tools that can be used to measure the performance of hardware-compiler combinations. The first is a standardized benchmark case with documented performance on 49 different hardware platforms, providing a quick way to compare performance across systems. This benchmark case is described in more detail in section 6.1. The second benchmarking tool is an automated benchmark suite that tests a broader range of MFC's functionality and summarizes the performance for each test in a single yaml file. The benchmarks are executed via `./mfc.sh bench --mem <gb/rank> -o <output>.yml -- -c <system> -n <nranks> <device_opts>`. The command line arguments are described in table 2.

The performance results for the benchmarks on different systems can be compared and summarized by running `./mfc.sh bench_diff <ref_ouput>.yml <output>.yml`, where `<ref_output>.yml` is the output file from running the benchmark suite on a reference system. After steps 1–4, the user can be confident that the hardware–compiler combination produces correct

Table 2: Arguments for the automated benchmarks.

Flag	Argument	Description
`--mem`	`<gb/rank>`	Memory (in GB) of problem size per rank
`-o`	`<output>`	Output yaml file with summary results
`-c`	`<system>`	Name of Mako template (Step 1)
`-n`	`<nranks>`	MPI ranks used for benchmarking
n/a	`<device_opts>`	`--gpu` (GPU); `--no-gpu` (CPU)

results and expected performance, and proceed to use MFC for their intended scientific use case.

Step 5: Running. With correctness and performance verified, the user can create a user-defined case file and run it with `./mfc.sh run`. The details for creating and running user-defined cases are beyond the scope of this work, but are described in detail in the MFC documentation[1].

4 Regression test details

The MFC regression suite tests over 500 unique cases (at the time of writing) and is designed to be readily extended and maintained by users. Each test case is based on a generic case file that can be modified using a stack mechanism to add or replace variables, enabling or disabling any MFC feature. Once a case is defined, an eight-digit universally unique identifier (UUID) is associated with it, and a directory in the source is created to store the case's golden file and associated metadata, including device information and build configuration, directly via CMake. The user can create the golden file and metadata by running the test with `./mfc.sh test -o <UUID> --generate`, where `<UUID>` is the unique identifier for the test case. More information on each of these steps is provided below.

4.1 Test case definition

Listing 2 demonstrates how a new codebase feature can be added to the MFC test suite. In this case, we call `alter_igr()` to create a span of tests that cover the relevant adorning features of MFC; 24 are made here by calling `alter_igr()` with six unique base stacks. Line 2 adds 'igr': 'T' and three other parameters to run all test cases that use it to the case stack, and adds 'IGR' to the human-readable trace that describes the case. Line 4 defines a loop over the available numerics added to the case stack and human-readable trace in line 5. Lines 6 and 7 define the test cases for two available iterative linear solvers via the function `define_case_d()` with the case stack, a human-readable trace entry, and a dictionary of additional variables to set in the case file. The contents of lines 2 and 5 are popped from the stack in lines 9 and 11 and return the stack to its original state. The stack-based approach enables ready extension and modification of the test suite without requiring user knowledge of other code features or their functionality. The human-readable trace appended by the function `alter_igr()` is printed to the command line, along with the case UUID, allowing the user to identify the case and its parameters.

---

[1] mflowcode.github.io/documentation

4.2 Golden file generation and maintenance

Golden files are reference output data used to verify the correctness of numerical simulations by comparing current test results against previously validated solutions. With MFC's toolchain, they are created, along with their associated metadata, by running the test case as `./mfc.sh test -o <UUID> --generate`, where `<UUID>` is the case identifier. This command executes the test, creating the golden file `golden.txt` and the metadata file `golden-metadata.txt`. These files use MFC's serial output formatting and record the CMake configuration, system information, and hardware information. Each line in `golden.txt` contains a flattened array storing a single simulation output. This format enables easy comparison of the output of different systems while minimizing the size of the golden files in version control. Once a golden file is created, test suite execution compares the code output to the golden file. It reports instances where an absolute or relative error exceeds a user-defined threshold. By default, an absolute and relative test tolerance of 1×10^{-12} is used for double precision computations. This tolerance reflects floating-point round-off and non-IEEE-754-compliant optimized floating-point operations at run time.

If test cases are changed so that other outputs must be added to the golden file, the user can update it. One updates golden files as `./mfc.sh test -o <UUID> --add-new-variables`. Executing this command adds new tracked variables to the golden file without modifying the existing values, thereby maintaining the integrity of the original data.

4.3 System and compiler bugs identified

The ability to build and test code that leverages a wide breadth of modern Fortran's features has enabled developers to identify and file tickets for at least 15 compiler and system bugs and regressions. The most common bugs identified are related to improper handling of module variables and unexpected behavior of the `!$acc routine seq` and `host_data use_device` directives. Once a bug is identified, a minimum working example can be created to open formal tickets with compiler vendors, aiding in their timely resolution. These bugs are relevant to core features of the Fortran language and OpenACC offloading, making their identification valuable to both MFC developers and the broader HPC community. Automated testing across core language features has highlighted idiosyncrasies among compilers using the same GPU offload model. The automated test suite has also proven invaluable in identifying bugs in the MFC source code itself, often due to refactoring that improves the overall quality of the source.

5 Performance test details

At the time of writing, MFC's automated benchmark suite contains five test cases that cover its most commonly used features. Test cases for new features can be added easily to the benchmark suite by creating a new directory with a case file in the `benchmarks/` directory and adding the case to the `toolchain/benchmarks.yml` file. Each benchmark case accepts an argument defining the approximate problem size per rank in gigabytes of memory and automatically scales to any number of MPI ranks. The summary data for each benchmark case is stored in a `yaml` file, which contains the total

Listing 2: Code snippet demonstrating how the stack-based approach adds a new code feature to the test suite.

```python
def alter_igr():
    stack.push('IGR',{'igr': 'T', 'alf_factor': 10, 'num_igr_iters': 10, 'num_igr_warm_start_iters': 10})

    for order in [3, 5]:
        stack.push(f"igr_order={order}", {'igr_order': order})
        cases.append(define_case_d(stack, 'Jacobi', {'igr_iter_solver': 1}))
        if order == 5:
            cases.append(define_case_d(stack, 'Gauss Seidel', {'igr_iter_solver': 2}))
        stack.pop()

    stack.pop()
```

wall time (in seconds), grindtime, and a summary of the invocation used to run the benchmark. The relative performance of two systems can be compared automatically using MFC's `bench_diff` tool, which prints a human-readable summary table. All benchmark cases use MFC's `--case-optimization` flag to specify certain case parameters as compile-time constants, enabling more aggressive compiler optimizations. The `--case-optimization` flag results in approximately a ten-fold improvement in grindtime performance, though speedup varies depending on the compiler and hardware used.

5.1 Performance bottlenecks identified

The automated benchmark suite has identified several performance regressions and bottlenecks, particularly on new architectures. The largest performance impacts occur when compilers cannot inline subroutines called within GPU compute regions. MFC uses negative indices to simplify array indexing when needed. When allocatable variables with non-default lower bounds are not defined at compile time, NVIDIA's NVHPC compiler does not mark them for inlining unless the compiler flag `-Minline=reshape` is specified, enabling speedups. Such subroutines are easily introduced when attempting to refactor code, so automatically identifying the resulting performance regression is valuable in maintaining MFC's performance. We observe additional inlining-related performance regressions with Cray CCE when using the OpenACC `!$acc routine seq` offloading directives. In some isolated cases, the CCE compiler chooses not to inline the subroutine unless the `!$acc routine seq` directive is replaced with the compiler hint `!$DIR INLINEALWAYS <routine_name>`. The instances in which the CCE `!$DIR INLINEALWAYS` hint is needed in place of the standard offload directive are not always obvious to developers, so automated benchmarking is a useful tool for identifying them.

Additional performance bottlenecks can be introduced in specific kernels where OpenACC struggles to produce efficient code. Such bottlenecks can be due to large register usage or compiler choices. For example, thread-private arrays that lack a known size at compile time require expensive memory reallocation for each independent loop when using the CCE compiler on AMD GPU devices. Reordering the contents of large kernels can also result in performance regressions if the resulting code uses registers less efficiently for thread-private variables. Due to its relatively small cache sizes, kernel ordering for efficient register use is especially relevant with AMD hardware. These kernel-specific performance bottlenecks can be challenging to identify when developing code. Automated benchmarking quickly identifies problems and provides a starting point for developers to determine the cause.

These performance regressions and bottlenecks identified by benchmarking demonstrate the value of automated benchmarks. Desirable yet straightforward changes, such as refactoring, can cause substantial performance regressions that are not apparent during development. MFC's automated benchmark suite offers a user-friendly approach to evaluating code performance and addressing identified regressions.

6 MFC as a tool for deployment and testing

MFC's portability and well-defined performance metrics make it useful for evaluating emerging supercomputers and hardware-software combinations. In the following sections, we present a standardized benchmark case with documented performance on nearly 50 different hardware platforms and strong and weak scaling results that serve as a reference for evaluating supercomputers.

6.1 A standardized benchmark case

Benchmark results are collected for a standardized three-dimensional (3D) CFD test problem. The test problem simulates a two-phase flow (such as gas and liquid interaction) using a well-established mathematical model entailing a system of eight coupled PDEs. The equations are solved using high-order numerical methods: fifth-order accurate WENO (weighted essentially non-oscillatory) spatial reconstructions for shock wave treatment, the HLLC (Harten–Lax–van Leer contact) Riemann solver for finite volume flux computation, and a third-order accurate Runge–Kutta method for time advancement. This combination of numerical methods is widely adopted in the CFD community for solving compressible and multiphase problems [2, 4, 5, 13, 27]. The benchmark case is maintained in MFC's version control under `examples/3D_performance_test/` and can be executed on any target system to evaluate performance.

Table 3 lists the grindtime performance of the standardized benchmark case run in double precision on a range of CPU, GPU,

Table 3: Observed grindtime (called "Time") performance (nanoseconds per grid cell, equation, and right-hand side evaluation) for a standardized compressible CFD test problem. Results are shown for various CPU, GPU, and APU architectures. The best performing compiler is shown in each case, with GNU, Intel, NVIDIA, AMD, and CCE tested as appropriate. Smaller numbers are better. All results are collected using the compiler that performs best for each hardware. CPU results are parallelized via MPI, with each rank bound to a single core. The results can be interpreted as providing relative performance between single GPUs and single CPU sockets.

Hardware	Type	Usage	Time	Hardware	Type	Usage	Time
NVIDIA GH200	APU	1 GPU	0.32	NVIDIA A10	GPU	1 GPU	4.3
NVIDIA H100 SXM5	GPU	1 GPU	0.38	AMD EPYC 7713	CPU	64 cores	5.0
NVIDIA H100 PCIe	GPU	1 GPU	0.45	Intel Xeon 8480CL	CPU	56 cores	5.0
AMD MI250X	GPU	1 GPU	0.55	Intel Xeon 6454S	CPU	32 cores	5.6
AMD MI300A	APU	1 APU	0.57	Intel Xeon 8462Y+	CPU	32 cores	6.2
NVIDIA A100	GPU	1 GPU	0.62	Intel Xeon 6548Y+	CPU	32 cores	6.6
NVIDIA V100	GPU	1 GPU	0.99	Intel Xeon 8352Y	CPU	32 cores	6.6
NVIDIA A30	GPU	1 GPU	1.1	Ampere Altra Q80-28	CPU	80 cores	6.8
AMD EPYC 9965	CPU	192 cores	1.2	AMD EPYC 7513	CPU	32 cores	7.4
AMD MI100	GPU	1 GPU	1.4	Intel Xeon 8268	CPU	24 cores	7.5
AMD EPYC 9755	CPU	128 cores	1.4	AMD EPYC 7452	CPU	32 cores	8.4
Intel Xeon 6980P	CPU	128 cores	1.4	NVIDIA T4	GPU	1 GPU	8.8
NVIDIA L40S	GPU	1 GPU	1.7	Intel Xeon 8160	CPU	24 cores	8.9
AMD EPYC 9654	CPU	96 cores	1.7	IBM Power10	CPU	24 cores	10
Intel Xeon 6960P	CPU	72 cores	1.7	AMD EPYC 7401	CPU	24 cores	10
NVIDIA P100	GPU	1 GPU	2.4	Intel Xeon 6226	CPU	12 cores	17
Intel Xeon 8592+	CPU	64 cores	2.6	Apple M1 Max	CPU	10 cores	20
Intel Xeon 6900E	CPU	192 cores	2.6	IBM Power9	CPU	20 cores	21
AMD EPYC 9534	CPU	64 cores	2.7	Cavium ThunderX2	CPU	32 cores	21
NVIDIA A40	GPU	1 GPU	3.3	Arm Cortex-A78AE	CPU	16 cores	25
Intel Xeon Max 9468	CPU	48 cores	3.5	Intel Xeon E5-2650V4	CPU	12 cores	27
NVIDIA Grace CPU	CPU	72 cores	3.7	Apple M2	CPU	8 cores	32
NVIDIA RTX6000	GPU	1 GPU	3.9	Intel Xeon E7-4850V3	CPU	14 cores	34
AMD EPYC 7763	CPU	64 cores	4.1	Fujitsu A64FX	CPU	48 cores	63
Intel Xeon 6740E	CPU	92 cores	4.2				

and APU architectures. We observe similar grindtimes when solving related problems, such as the inviscid Euler equations (4 PDEs) and the six-equation multiphase flow model [26] (10 PDEs). We report grindtimes using the compiler that produced the best results for each system. Current tested compilers include Cray's CCE, NVIDIA's NVHPC, AMD's AOCC, Intel's OneAPI, and GNU GCC. We benchmark nominally single-precision GPUs in double precision using hardware or compiler conversion. Results for GPUs with more than one compute die (for example, the AMD MI250X and MI300A) are presented for the entire device. Parallelism is achieved using MPI; for CPUs, each MPI rank is bound to a core. CPUs may have more cores than the results reported for. However, results are reported for the core count providing optimal performance. Additional details on the hardware, compiler, and systems used are available in the MFC documentation[2]. This collection of results is a reference for users to compare the performance of their hardware–software combinations against a range of systems spanning several generations.

6.2 Weak scaling

Weak scaling tests in MFC are performed using the same standardized case described in section 6.1 with modified discretization and domain boundaries. The domain boundaries and associated discretization are selected so that each MPI rank holds a local domain with a perfect cube of grid cells. Uniform local discretization ensures uniform communication costs. Table 4 lists example MPI discretizations and problem sizes used to collect the weak scaling results for OLCF Frontier in fig. 2. The Frontier weak scaling test is conducted for a problem size of 200^3 grid cells per MI250X GCD, which amounts to about 16 GB of HBM2e memory. The size is selected to be large enough to saturate the memory bandwidth of the MI250X GCDs to reach MFC's peak performance. Weak scaling performance is measured using the previously defined grindtime metric. The grindtime multiplied by the number of ranks should remain constant across problem sizes for ideal weak scaling, which we observe in all cases. The file-per-process I/O strategy described in [37] reduces I/O overhead in weak scaling tests. This approach is used when the number of MPI ranks exceeds 10^4 or the total problem size exceeds 100 billion spatially discretized grid cells.

---

[2]mflowcode.github.io/documentation/md_expectedPerformance.html

Table 4: MPI decomposition and discretization details for a weak scaling test on OLCF Frontier. Each MPI rank has a local domain of $200 \times 200 \times 200$ grid cells so that all halo exchanges are equivalent. Approximately 16 GB of HBM2e memory is used per MI250X GCD, or one quarter of the available HBM2e memory.

# Ranks	Decomposition	Discretization	# Cells [B]
128	$4 \times 4 \times 8$	$800 \times 800 \times 1600$	1.02
384	$6 \times 8 \times 8$	$1200 \times 1600 \times 1600$	3.07
1024	$8 \times 8 \times 16$	$1600 \times 1600 \times 3200$	8.19
3072	$12 \times 16 \times 16$	$2400 \times 3200 \times 3200$	24.6
8192	$16 \times 16 \times 32$	$3200 \times 3200 \times 6400$	65.5
24576	$24 \times 32 \times 32$	$4800 \times 6400 \times 6400$	197
65536	$32 \times 32 \times 64$	$6400 \times 6400 \times 12800$	524

Table 5: Weak scaling efficiencies and device counts for the systems shown in fig. 2. Near-ideal efficiency is observed over three orders of magnitude in problem size for all systems across multiple generations of compute devices from NVIDIA and AMD.

System	Base case	Limit case	Efficiency
OLCF Summit	216 GPUs	13825 GPUs	97%
CSCS Alps	64 GPUs	9200 GPUs	97%
OLCF Frontier	128 GCDs	65536 GCDs	95%
LLNL El Capitan	64 GPUs	32768 GPUs	99%

Figure 2 shows weak scaling results for MFC on four leadership-class supercomputers. Three of the systems shown (OLCF Summit [30], OLCF Frontier [33], and LLNL El Capitan [32]) have held the number one position on the TOP500 list, and CSCS Alps [31] is in the top ten and is the largest Grace Hopper-based machine online at the time of writing. We observe weak scaling efficiencies above 95% for all systems, spanning three orders of magnitude in problem size and scaling to full systems. Table 5 shows each system's base case, limit case, and efficiency. MFC's consistent scaling performance across these systems makes it a suitable tool for testing weak scaling performance.

6.3 Strong scaling

Scaling tests in MFC are performed using the standardized case of section 6.1 with different spatial grid discretizations. Performance is measured using the previously defined grindtime, which should scale inversely with the number of processors used for a constant problem size. The problem size for the base case is selected to saturate the available GPU memory of 8 MPI ranks. Saturating the device memory minimizes relative communication cost in the base case. Using 8 MPI ranks enables uniform MPI communication across the three spatial directions. The maximum problem size per GCD on OLCF Frontier is approximately 32 M grid cells. So, an initial problem size of $634 \times 634 \times 634$ is used for scaling tests. This results in 31.9 M grid cells per device in the 8 rank base case. MFC supports GPU-Aware MPI (via RDMA), which is not enabled by default. It
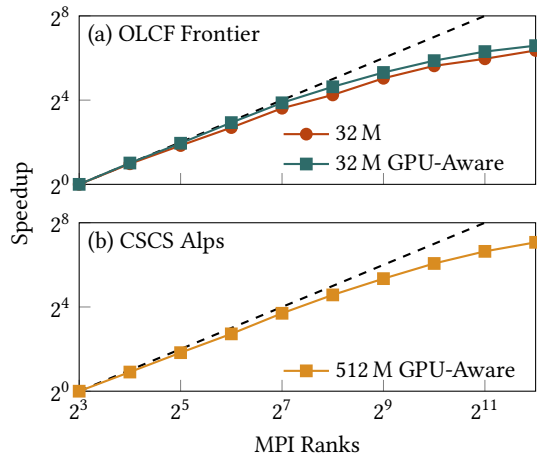

Figure 2: Weak scaling results for MFC on five flagship supercomputers. Near-ideal scaling is observed for multiple generations of AMD and NVIDIA hardware. Table 5 shows the details of each system's base case, limit case, and efficiency.

can be enabled by adding `'rdma_mpi': 'T'` to the input case file when the machine supports it. Figure 3 (a) shows that GPU-aware MPI improves strong scaling efficiency on Frontier.

The strong scaling results for CSCS Alps in fig. 3 (b) use the alternative numerics described in [36] that enable use of a larger base case. The larger base case is discretized using a 1600^3 spatial grid, which results in 512 M grid cells per device in the 8 rank base case. The larger base case shows preferable scaling efficiency on CSCS Alps. However, the trend is similar to that observed on OLCF Frontier. MFC's predictable scaling performance trends make it a valuable tool for evaluating the network performance of new supercomputers.

7 Limitations of current work

MFC's suitability as a tool for testing and benchmarking supercomputers has limitations. The primary limitation is the reliance on compiler support for directives, either OpenMP or OpenACC, for offloading to non-CPU devices. Currently, OpenMP and OpenACC provide limited or no support for more esoteric computing devices,

Figure 3: Strong scaling performance on (a) OLCF Frontier and (b) CSCS Alps. The speedup is calculated as the ratio of the grindtime for a given processor count to the grindtime of the 8 rank baseline. The impact of using GPU-Aware MPI to reduce communication overhead and improve strong scaling efficiency is shown in the OLCF Frontier results. Extension of near-ideal strong scaling behavior follows from using a larger base case on CSCS Alps.

such as Cerebras wafer-scale engines, Graphcore IPUs, NextSilicon Maverick-2, or quantum computers. Researchers are starting to use these devices for stencil computations similar to those in MFC [6, 18, 28], though the current implementations rely on specialized compilers and lack portability. Until these architectures are widely adopted and garner mainstream compiler support, MFC will be limited to testing CPU, GPU, and APU devices from major vendors.

8 Conclusion

User-friendly applications are essential for evaluating the real-world performance of emerging supercomputers. MFC's portability, well-defined performance metrics, and automated testing suite make it suitable for such evaluations. Any user can run automated testing and benchmarking after a straightforward setup process. This process abstracts away the details of the hardware and system. The automated tools of MFC have been used to identify and report over 15 compiler and system bugs. The results have identified numerous performance regressions when systems are updated. MFC has also shown high-quality scaling performance across multiple generations of GPU hardware from NVIDIA and AMD. These data serve as a reference for evaluating the performance of new computers. These features and results establish MFC as a useful tool that nearly any user can use to test and benchmark supercomputers or incorporate into existing benchmarking suites.

Acknowledgments

Code Availability

MFC is an open-source project; it is available under the MIT license. The MFC source code is available at `https://github.com/MFlowCode/MFC`. Additional information, documentation, and example simulations are available at `https://mflowcode.github.io`.

References

[1] Advanced Micro Devices Inc. 2020. hipFFT: High-Performance Fast Fourier Transform Library. `https://github.com/ROCmSoftwarePlatform/hipFFT` GitHub Repository.

[2] Antonis F. Antoniadis, Dimitris Drikakis, Pericles S. Farmakis, Lin Fu, Ioannis Kokkinakis, Xesús Nogueira, Paulo A.S.F. Silva, Martin Skote, Vladimir Titarev, and Panagiotis Tsoutsanis. 2022. UCNS3D: An open-source high-order finite-volume unstructured CFD solver. *Comput. Phys. Commun.* 279 (2022), 108453.

[3] Mike Bayer. 2024. *Mako: A Python template library.* `https://www.makotemplates.org/` Version 1.3.3, Accessed: 2025-07-18.

[4] Matteo Bernardini, Davide Modesti, Francesco Salvadore, and Sergio Pirozzoli. 2021. STREAmS: A high-fidelity accelerated solver for direct numerical simulation of compressible turbulent flows. *Comput. Phys. Commun.* 263 (2021), 107906.

[5] Deniz A. Bezgin, Aaron B. Buhendwa, and Nikolaus A. Adams. 2023. JAX-Fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows. *Comput. Phys. Commun.* 282 (2023), 108527.

[6] Nick Brown, Brandon Echols, Justs Zarins, and Tobias Grosser. 2022. Exploring the Suitability of the Cerebras Wafer Scale Engine for Stencil-Based Computation Codes. In *Euro-Par 2022: Parallel Processing Workshops: Euro-Par 2022 International Workshops, Glasgow, UK, August 22–26, 2022, Revised Selected Papers* (Glasgow, United Kingdom). Springer-Verlag, Berlin, Heidelberg, 51–65.

[7] Spencer H. Bryngelson and Tim Colonius. 2020. Simulation of humpback whale bubble-net feeding models. *J. Acoust. Soc. Am.* 147, 2 (2020), 1126–1135.

[8] Spencer H. Bryngelson, Rodney O. Fox, and Tim Colonius. 2023. Conditional moment methods for polydisperse cavitating flows. *J. Comput. Phys.* 477 (2023), 111917.

[9] Spencer H. Bryngelson, Kevin Schmidmayer, Vedran Coralic, Jomela C. Meng, Kazuki Maeda, and Tim Colonius. 2021. MFC: An open-source high-order multi-component, multi-phase, and multi-scale compressible flow solver. *Comput. Phys. Commun.* 266 (2021), 107396.

[10] A. Charalampopoulos, S. H. Bryngelson, T. Colonius, and T. P. Sapsis. 2022. Hybrid quadrature moment method for accurate and stable representation of non-Gaussian processes and their dynamics. *Phil. Trans. R. Soc. A* 380, 2229 (2022).

[11] Esteban Cisneros-Garibay, Henry Le Berre, Spencer H. Bryngelson, and Jonathan B. Freund. 2025. Pyrometheus: Symbolic abstractions for XPU and automatically differentiated computation of combustion kinetics and thermodynamics. *arXiv preprint arXiv:2503.24286* (2025).

[12] CSCS. 2025. cscs-reframe-tests. `https://github.com/eth-cscs/cscs-reframe-tests` GitHub Repository.

[13] Francesco De Vanna, Filippo Avanzi, Michele Cogo, Simone Sandrin, Matt Bettencourt, Francesco Picano, and Ernesto Benini. 2023. URANOS: A GPU accelerated Navier-Stokes solver for compressible wall-bounded flows. *Comput. Phys. Commun.* 287 (2023), 108717.

[14] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience* 15, 9 (2003), 803–820.

[15] Matteo Frigo and Steven G. Johnson. 2005. The design and implementation of FFTW3. *Proc. IEEE* 93, 2 (2005), 216–231.

[16] Douglas Jacobsen and Robert F Bird. 2023. Ramble: A flexible, extensible, and composable experimentation framework. In *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis* (Denver, CO, USA) *(SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 600–608.

[17] Junjie Li, Alexander Bobyr, Swen Boehm, William Brantley, Holger Brunst, Aurelien Cavelan, Sunita Chandrasekaran, Jimmy Cheng, Florina M. Ciorba, Mathew Colgrove, Tony Curtis, Christopher Daley, Mauricio Ferrato, Mayara Gimenes de Souza, Nick Hagerty, Robert Henschel, Guido Juckeland, Jeffrey Kelling, Kelvin Li, Ron Lieberman, Kevin McMahon, Egor Melnichenko, Mohamed Ayoub Neggaz, Hiroshi Ono, Carl Ponder, Dave Raddatz, Severin Schueller, Robert Searles, Fedor Vasilev, Veronica Melesse Vergara, Bo Wang, Bert Wesarg, Sandra Wienke, and Miguel Zavala. 2022. SPEChpc 2021 Benchmark Suites for Modern HPC Systems. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering* (Bejing, China) *(ICPE '22)*. Association for Computing Machinery, New York, NY, USA, 15–16.

[18] Thorben Louw and Simon McItosh-Smith. 2021. Using the Graphcore IPU for traditional HPC applications. In *3rd Workshop on Accelerated Machine Learning (AccML)*.

[19] Sebastian Lührs, Daniel Rohe, Alexander Schnurpfeil, Kay Thust, and Wolfgang Frings. 2016. Flexible and Generic Workflow Management. In *Parallel Computing: On the Road to Exascale* (2015-09-01) *(Advances in parallel computing, Vol. 27)*. International Conference on Parallel Computing 2015, Edinburgh (United Kingdom), 1 Sep 2015 - 4 Sep 2015, IOS Press, Amsterdam, 431–438.

[20] Robert McLay, Kirk W. Schulz, William L. Barth, and Todd Minyard. 2011. Best practices for the deployment and management of production HPC clusters. In *State of the Practice Reports, SC '11*. ACM, New York, NY, USA, 9:1–9:11.

[21] NVIDIA Corporation. 2007. cuFFT: High-Performance CUDA FFT Library.

[22] OLCF. 2025. olcf-test-harness. https://github.com/olcf/olcf-test-harness GitHub Repository.

[23] OpenMP Architecture Review Board. 2021. *OpenMP Application Programming Interface Version 5.2*. OpenMP Architecture Review Board. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf.

[24] Achyut Panchal, Spencer H. Bryngelson, and Suresh Menon. 2023. A seven-equation diffused interface method for resolved multiphase flows. *J. Comput. Phys.* 475 (2023), 111870.

[25] Anand Radhakrishnan, Henry Le Berre, Benjamin Wilfong, Jean-Sebastien Spratt, Mauro Rodriguez Jr., Tim Colonius, and Spencer H. Bryngelson. 2024. Method for portable, scalable, and performant GPU-accelerated simulation of multiphase compressible flow. *Comput. Phys. Commun.* 302 (2024), 109238.

[26] Richard Saurel, Fabien Petitpas, and Ray A. Berry. 2009. Simple and efficient relaxation methods for interfaces separating compressible fluids, cavitating flows and shocks in multiphase mixtures. *J. Comput. Phys.* 228, 5 (2009), 1678–1712.

[27] Kevin Schmidmayer, Fabien Petitpas, Sébastien Le Martelot, and Éric Daniel. 2020. ECOGEN: An open-source tool for multiphase, compressible, multiphysics flows. *Computer Physics Communications* 251 (2020), 107093.

[28] Z. Song, R. Deaton, B. Gard, and S. H. Bryngelson. 2025. Incompressible Navier–Stokes solve on noisy quantum hardware via a hybrid quantum–classical scheme. *Computers & Fluids* 288 (2025), 106507.

[29] TACC. 2025. benchpro. https://github.com/TACC/benchpro GitHub Repository.

[30] TOP500.org. 2024. Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband. https://www.top500.org/system/179397/. Accessed: 2025-07-18.

[31] TOP500.org. 2025. Alps - HPE Cray EX254n, NVIDIA Grace 72C 3.1GHz, NVIDIA GH200 Superchip, Slingshot-11, HPE Cray OS. https://www.top500.org/system/180259/. Accessed: 2025-07-18.

[32] TOP500.org. 2025. El Capitan – HPE Cray EX255a, AMD 4th Gen EPYC 24C 1.8GHz, AMD Instinct MI300A, Slingshot-11, TOSS. https://www.top500.org/system/180307/. Accessed: 2025-07-18.

[33] TOP500.org. 2025. Frontier – HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE Cray OS. https://www.top500.org/system/180047/. Accessed: 2025-07-18.

[34] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter Mey. 2012. OpenACC — First Experiences with Real-World Applications. In *Euro-Par 2012 Parallel Processing*, Christos Kaklamanis, Theodore Papatheodorou, and Paul G. Spirakis (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 859–870.

[35] Benjamin Wilfong, Henry Le Berre, Anand Radhakrishnan, Ansh Gupta, Diego Vaca-Revelo, Dimitrios Adam, Haocheng Yu, Hyeoksu Lee, Jose Rodolfo Chreim, Mirelys Carcana Barbosa, Yanjun Zhang, Esteban Cisneros-Garibay, Aswin Gnanaskandan, Mauro Rodriguez Jr., Reuben D. Budiardja, Stephen Abbott, Tim Colonius, and Spencer H. Bryngelson. 2025. MFC 5.0: An exascale many-physics flow solver. *arXiv preprint arXiv:2503.07953* (2025).

[36] Benjamin Wilfong, Anand Radhakrishnan, Henry Le Berre, Daniel J. Vickers, Tanush Prathi, Nikolaos Tselepidis, Benedikt Dorschner, Reuben Budiardja, Brian Cornille, Stephen Abbott, Florian Schäfer, and Spencer H. Bryngelson. 2025. Simulating many-engine spacecraft: Exceeding 1 quadrillion degrees of freedom via information geometric regularization. *arXiv preprint arXiv:2505.07392* (2025).

[37] Benjamin Wilfong, Anand Radhakrishnan, Henry A. Le Berre, Stephen Abbott, Reuben. D. Budiardja, and Spencer H. Bryngelson. 2024. OpenACC offloading of the MFC compressible multiphase flow solver on AMD and NVIDIA GPUs. In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, Atlanta, Georgia, USA, 1923–1933.