Can Fortran's 'do concurrent' Replace Directives for Accelerated Computing?

Miko M. Stulajter[(✉)], Ronald M. Caplan,
and Jon A. Linker

Predictive Science Inc.,
9990 Mesa Rim Road Suite 170,
San Diego, CA 92121, USA
{miko,caplanr,linkerj}@predsci.com
http://www.predsci.com

Abstract. Recently, there has been growing interest in using standard language constructs (e.g. C++'s Parallel Algorithms and Fortran's `do concurrent`) for accelerated computing as an alternative to directive-based APIs (e.g. OpenMP and OpenACC). These constructs have the potential to be more portable, and some compilers already (or have plans to) support such standards. Here, we look at the current capabilities, portability, and performance of replacing directives with Fortran's `do concurrent` using a mini-app that currently implements OpenACC for GPU-acceleration and OpenMP for multi-core CPU parallelism. We replace as many directives as possible with `do concurrent`, testing various configurations and compiler options within three major compilers: GNU's `gfortran`, NVIDIA's `nvfortran`, and Intel's `ifort`. We find that with the right compiler versions and flags, many directives can be replaced without loss of performance or portability, and, in the case of `nvfortran`, they can all be replaced. We discuss limitations that may apply to more complicated codes and future language additions that may mitigate them. The software and Singularity/Apptainer containers are publicly provided to allow the results to be reproduced.

Keywords: accelerated computing · OpenMP · OpenACC · do concurrent · standard language parallelism

1 Introduction

OpenMP[1] [11] and OpenACC[2] [5] are popular directive-based APIs for parallelizing code to run on multi-core CPUs and GPUs. For accelerated computing,

---

[1] www.openmp.org.

[2] www.openacc.org.

---

they provide a higher-level approach to accelerating codes without requiring writing specialized low-level, often vendor-specific, API code (e.g. CUDA, ROCm, OpenCL, etc.). Since they mostly consist of specialized comments/pragmas, they exhibit backward compatibility, allowing a non-supported compiler to simply ignore them and still compile the code as before. This makes directive-based approaches very desirable for legacy codes, and helps to allow compartmentalized development. However, they also can suffer from incomplete vendor, hardware, and/or compiler support, make codes somewhat harder to read, and, due to their rapid development, are less future-proof than standard languages, possibly requiring occasional re-writes.

Due to the widespread adoption of multi-core CPUs and accelerators, standard languages have begun to add built-in features that may help/enable compilers to parallelize code. This includes C++17's Standard Parallel Algorithms and Fortran's `do concurrent` (DC) (see Refs. [6,10] for examples using the NVIDIA HPC SDK[3]). Standard parallel language features have the potential to remove the need for directives, making multi-threaded and accelerated codes fully portable across compiler vendors and hardware. However, this requires compiler support, and few have been quick to implement these features for GPU acceleration.

Here, we focus on Fortran's DC construct. The NVIDIA HPC SDK is the only compiler at the time of this writing with accelerator support using DC, while Intel has indicated plans to add such support in an upcoming release of their `ifort` compiler included in the OneAPI HPC Toolkit [13]. Other compilers that support directive-based accelerator offloading in Fortran include GCC's `gfortran`[4], LLVM `flang`[5], AOCC's extended `flang`[6], IBM's `XL`[7], and HPE's Cray Fortran[8], but we could not find any announced plans for these to support DC for accelerated computing in the near future[9].

In this paper, we investigate the current capabilities, portability, and performance of replacing directives with DC in a Fortran mini-app that currently implements directives for GPU-acceleration and multi-core CPU parallelism. We replace as many directives as possible with DC, testing various run-time configurations and compilers. Our mini-app currently uses OpenACC with either `nvfortran` or `gfortran` for GPU-acceleration on NVIDIA GPUs, and uses OpenMP with `nvfortran`. `gfortran`, or `ifort` for multi-core CPU parallelism (as well as OpenACC multi-core with `nvfortran`). A key portability concern is if replacing directives with DC for GPU-acceleration will result in a loss of multi-core CPU parallelism. Therefore, we test if each compiler can parallelize

---

[3] https://developer.nvidia.com/hpc-sdk.

[4] https://gcc.gnu.org/.

[5] https://flang.llvm.org.

[6] https://developer.amd.com/amd-aocc.

[7] https://www.ibm.com/products/xl-fortran-linux-compiler-power.

[8] https://support.hpe.com/hpesc/public/docDisplay?docId=a00115296en_us&page=index.html.

[9] Note that as this paper was going to press, HPE has indicated plans to support DC on GPUs.

the DC loops for multi-core CPUs. We note that for codes using non-hybrid MPI for CPU parallelism and MPI+OpenMP/ACC for GPU acceleration, this is not as much of a concern.

The paper is organized as follows: In Sect. 2, we describe our Fortran mini-app with its current directive-based parallelelization, along with the test case we use, showing baseline performance results. In Sect. 3, we describe the implementation of DC into the mini-app, first introducing its capabilities and support, and then showing examples of replacing OpenMP/ACC directives with DC, including a discussion of current limitations. Then the resulting mini-app source code versions and compiler flag options used for the tests are described. Performance and compatibility results are reported in Sect. 4 for both multi-core CPU and GPU runs. Finally, discussion on the current status of DC and its potential to replace directives is given in Sect. 5. Instructions on how to access and use our provided Singularity/Apptainer containers and codes to reproduce the results in the paper are given in the Appendix.

2   Code and Test Description

To investigate the current capabilities, portability, and performance of replacing directives with DC, we use a Fortran mini-app called `diffuse` that currently implements directives for GPU-acceleration and multi-core CPU parallelism. Here we describe the code, the test case we use, the computational test environment, and baseline performance results.

2.1   Code Description

NASA and NSF have recently supported a program called "Next Generation Software for Data-driven Models of Space Weather with Quantified Uncertainties", whose main objective is to improve predictions of solar wind and coronal mass ejections to investigate how they might impact Earth. This will be done by developing a new data-driven time-dependent model of the Sun's upper atmosphere. One key component of this model is the use of a data-assimilation flux transport model to generate an ensemble of magnetic field maps of the solar surface to use as boundary conditions. To accomplish this, we have been developing an Open-source Flux Transport (OFT) software suite, whose key computational core is the High-Performance Flux Transport code (HipFT). HipFT currently implements OpenACC for GPU-acceleration and OpenMP for multi-core CPU parallelism, and we are interested in replacing the directives with DC.

In order to test the use of DC, we use a mini-app called `diffuse` that implements the most computationally expensive algorithm (surface diffusion) of the flux transport in HipFT. `diffuse`'s source code for the diffusion algorithm is identical to that of HipFT. The diffusion algorithm integrates a spherical surface heat equation on a logically rectangular non-uniform grid. The operator is discretized with a second-order central finite-difference scheme in space, while the time integration uses the second-order Legendre polynomial extended stability Runge-Kutta scheme (RKL2) [3,8].

2.2   Test Description

Although `diffuse` is used here as a mini-app representation of HipFT, it is also used in production to slightly smooth solar surface magnetic fields to prepare them for use in models of the corona [2]. As such, we select a real-world example of using `diffuse`, that of smoothing the 'Native res PSI map' described in Ref. [4]. This large map has a resolution of 3974×2013 in (θ, ϕ) and takes 40,260 total iterations of applying the diffusion operator to smooth. A detail from the map before and after running `diffuse` is shown in Fig. 1.
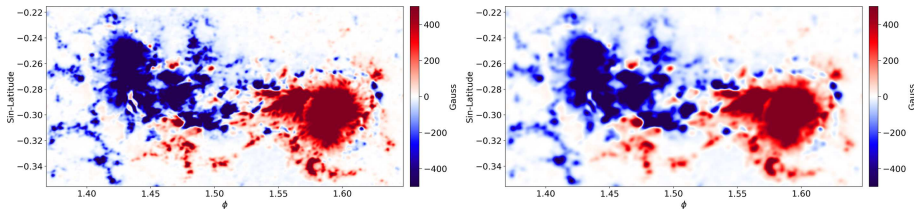

Fig. 1. Zoomed-in detail of the test case magnetic field map before (left) and after (right) smoothing with `diffuse`.

2.3   Computational Environment

In order to best assess the capabilities of the compiler support for DC, we use the latest available versions of the compilers at the time of testing. These are shown in Table 1. The CPU tests are run on the Bridges2 system located at the Pittsburgh Supercomputing Center using our allocation obtained through NSF's XSEDE program [12]. The GPU tests are run on an NVIDIA DGX A100 server at San Diego State University. Since `diffuse` does not have multi-node or multi-GPU capabilities, the CPU tests are run on a single CPU node, while the GPU tests are run on a single GPU within the DGX system. The hardware specifications are shown in Table 2.

Table 1. Compiler versions used in tests.

Compiler Suite	Compiler	Version
GNU Compiler Collection	`gfortran`	11.2
NVIDIA HPC SDK	`nvfortran`	21.7
Intel OneAPI HPC Toolkit	`ifort` (classic)	21.3

Since systems do not always have the latest compilers available, and setting up our code's dependencies can be difficult, we utilize Singularity/Apptainer containers [7]. These containers are built with the compiler environment and our dependent libraries pre-installed so they can be easily used to build and

Table 2. Hardware utilized for all test runs.

	CPU	GPU
CPU/GPU Model	(2x) AMD EPYC 7742	NVIDIA A100
	(128 cores)	SXM4
Peak Memory Bandwidth	381.4 GB/s	1555 GB/s
Clock Frequency (base/boost)	2.3/3.4 GHz	1.1/1.4 GHz
RAM	256 GB	40 GB
Peak DP FLOPs	7.0 TFLOPs	9.8 TFLOPs

run the code. We use Singularity 3.8.0, and for GPU runs, use the '--nv' flag to connect to the NVIDIA driver (and CUDA library) on the local system. The CUDA run-time library used for GPU runs was version 11.4. As shown in the Appendix, running the codes in the containers yields virtually the same performance as a bare metal installation. All the test runs performed in this paper can be reproduced using the containers along with the code, both of which are publicly released in Ref. [9] and at www.predsci.com/papers/dc.

2.4   Baseline Performance Results

Our goal in this paper is to test replacing directives with DC for accelerated computing, ensuring we do not lose multi-core CPU parallelism, and that the performance is comparable to the original directive-based code. It is not our focus to compare performance between the various compilers and hardware. We therefore use similar basic compiler optimization flags (shown in Sect.3.2) for each compiler-hardware combination and do not explore all possible optimizations. In order to compare the performance of the original code to the modified versions, we perform baseline timings of the original code. For these, and all timing results in the paper, we run each test 10 times and take the average of the full wall clock times (which include all I/O and GPU-CPU data transfer time). In Fig. 2 we show the baseline timings along with their standard deviations. We also include CPU runs on a single CPU core (serial) to illustrate the multi-core CPU parallelism. We see that each compiler obtains comparable performance on the CPU runs, yielding a speedup of $\sim 7\times$ when using 128 CPU cores compared to running in serial. While this may seem low, it is common for highly memory-bound algorithms to exhibit such non-ideal single node multi-threaded scaling [1]. The performance of the `nvfortran` CPU run using OpenMP is $\sim 10\%$ faster than using OpenACC for multi-core parallelism. On the GPU, the `nvfortran` OpenACC GPU run is $\sim 30\%$ faster than the `gfortran` run, which is not unexpected considering `nvfortran` has a more mature implementation of OpenACC.
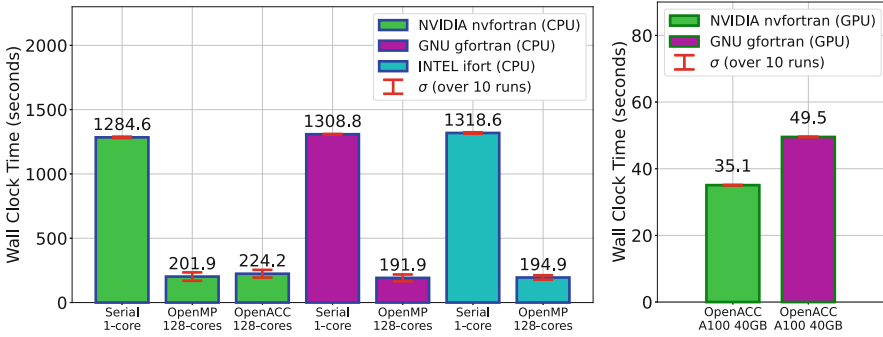
Fig. 2. Baseline CPU and GPU timing results of the original `diffuse` code run on the test case. Times shown are averages over 10 runs, and the standard deviations are shown.

3   Implementation

In this section, we first give a background on Fortran's `do concurrent` construct, and then describe our implementation of it into the `diffuse` code, and the resulting code variations. We also describe the compiler flags used for each code version and compiler combination.

3.1   The Fortran `do concurrent` construct

In 2008, ISO Standard Fortran introduced the DC construct for loops as an alternative to the standard `do` loop (or nested `do` loops). DC indicates to the compiler that the loop's iterations can be computed in any order. This potentially allows for the expression of parallelism of loops directly in the Fortran language, making it easier for compilers to parallelize the loops. While any-order execution is a necessary condition for parallelization, it is not always sufficient (for example, reduction and atomic operations, as well as others[10]). Therefore, DC can be viewed as providing a hint to the compiler that the loop is likely able to be parallelized. Work in helping make DC fully sufficient for parallelism is on-going, with Fortran 2018 adding locality statements (allowing specification of private and shared variables[11]), and specifying reductions in DC will be included in the upcoming Fortran 202X release[12].

The syntax of non-nested `do` loops and `do concurrent` loops are similar. A `do` loop has the syntax `do index=start,end` while a `do concurrent` loop has the syntax `do concurrent (index=start:end)`. The only key difference is the addition of the word `concurrent` and a small change to the loop parameters where there is the addition of parentheses and a replacement of the comma with

---

[10] https://releases.llvm.org/12.0.0/tools/flang/docs/DoConcurrent.html.

[11] https://j3-fortran.org/doc/year/18/18-007r1.pdf.

[12] https://j3-fortran.org/doc/year/21/21-007.pdf.

ellipses. With nested loops, there is more of a difference in formatting. Code 1 shows nested do loops parallelized with directives. The loop nest is shown with both OpenMP and OpenACC directives in the manner they are used in the mini-app. This nested do loop example spans 8 lines with directives, but can be written in 3 lines with DC as shown in Code 2. With DC loops, nested loops initialization statements are collapsed into one initialization statement. The syntax of DC loops is as follows: do concurrent (index1=start1:end1, index2=start2:end2, ...). As this example shows, DC loops make nested do loops more compact and easier to read.

Code 1. Nested do loops with OpenMP/ACC directives

```
!$omp parallel do collapse(2) default(shared)
!$acc parallel loop collapse(2) default(present)
      do  i=1,N
          do  j=1,M
              Computation
          enddo
      enddo
!$acc end parallel loop
!$omp end parallel do
```

Code 2. Nested do loops as a do concurrent loop

```
      do concurrent  (i=1:N, j=1:M)
          Computation
      enddo
```

Most current compilers support the Fortran 2008 standard, which includes the basic DC syntax. However, since the specification does not require that the compiler try to parallelize the loops, they are often treated as serial do loops. When a compiler does support parallelization of DC, special compiler flags are needed to activate the feature (see Sect. 3.3 for details).

Although the latest version of the OpenACC (3.1)[13] specification adds support for decorating DC loops with directives, at present, there are no implementations of this support (with the possible exception of using the kernels directive). There is also no mention of supporting directives on a DC construct within the most recent OpenMP (5.1)[14] specification. Therefore, replacing do loops with DC may break the ability to parallelize the loops when using compilers that do not support direct DC parallelization.

The current state of DC support is varied across different compilers and versions. nvfortran 18.1 added serial support for DC along with locality of variables, while nvfortran 20.11 added support for parallelization of DC loops

---

[13] https://www.openacc.org/blog/announcing-openacc-31.
[14] https://www.openmp.org/spec-html/5.1/openmp.html.

for both CPUs and GPUs. In `gfortran` 8, serial support for DC was introduced, while `gfortran` 9 added support for parallelization of DC on multi-core CPUs using `gfortran`'s auto parallelization feature. `ifort` started supporting serial DC loops in version 12. Then, in version 16, parallelization support was added through `ifort`'s auto parallelization feature (using the flag `-parallel`). With version 19.1, locality of variable support was added, and parallelization became linked to the OpenMP compiler flags. Table 3 gives a summary of the current support of parallel DC loops for the compilers used in this paper.

Table 3. Current support of DC loop parallelization for the compilers used in this paper.

Compiler	Version	`do concurrent` parallelization support
`gfortran`	≥9	Parallelizable on CPU with `-ftree-parallelize-loops=<N>` flag. Locality of variables is *not* supported.
`nvfortran`	≥20.11	Parallelizable on CPU and GPU with the `-stdpar` flag. Locality of variables is supported.
`ifort`	≥19.1	Parallelizable on CPU with the `-fopenmp` flag. Locality of variables is supported

3.2 Code Versions

Here we list the code variants that we use to test the portability and performance of replacing directives with DC in `diffuse`. For versions that use DC, only basic DC loop syntax was used with no locality of variables, as not all compilers support this feature in all configurations.

Original: This is the original version of `diffuse` which uses OpenACC and OpenMP directives on all parallelizable `do` loops as well as OpenACC data movement directives. It does not contain any DC loops. It is the code version used for the performance results of Sect. 2.4, and will be the standard we compare to for both performance and compatibility.

New: This version is obtained by replacing directive surrounded `do` loops in *Original* with DC loops, with the exception of reduction loops. The directives on the reduction loops are kept since reductions are not supported in parallelized DC loops (see discussion in *Experimental*). We also keep all OpenACC data directives for explicit GPU data management. This code is expected to perform as well as the *Original* code if the DC loops are recognized and implemented efficiently.

Serial: This version contains no OpenACC or OpenMP directives at all, nor any DC loops. It is the same as *Original* with all directives removed. It should run in serial in all cases, unless an auto-parallelizing feature of a compiler is utilized.

We include this code as a control and to ensure the multi-core CPU parallel runs are exhibiting the expected parallelism.

Experimental: This version does not contain any OpenMP or OpenACC directives at all, replacing all loops (including reduction loops) with DC. A key feature of this code version is that it represents the 'ideal' scenario of using only the Fortran standard language for accelerated computing without needing any directives. This version does not technically violate the Fortran standard since a DC on a reduction loop is valid if not parallelized, as the iterations can be computed in any order. However, if the compiler does parallelize these DC reduction loops, it will likely produce wrong results due to race conditions, unless it supports implicit analysis and implementation of DC reductions. As mentioned above, Fortran 202X will add reductions to DC, resolving this problem. Removing all directives also removes explicit GPU-CPU data movement, whose absence will lead to very poor performance on accelerators (due to repeated data movement between the CPU and GPU) unless the compiler supports automatic GPU-CPU memory management. Features such as NVIDIA's Unified Memory and AMD's Smart Access Memory can allow compilers to resolve this issue.

In Table 4 we summarize all versions of the code we use for our tests.

Table 4. Summary of DC and directive implementations for each version of the `diffuse` code tested.

	do concurrent	Directives
Original	None	all loops & data management
New	all loops except reductions	reduction loops & data management
Serial	None	None
Experimental	all loops	None

3.3   Compiler Flag Options

The `gfortran`, `nvfortran`, and `ifort` compilers each have different flags to implement code parallelization and optimizations. Here we describe the compiler flags we use for each code version, compiler, and target hardware configuration. For all compilers, we use the `-O3` flag to activate typical compiler optimizations, and `-march=<ARCH>` to tell the compiler to target the specific CPU we run the tests on. Typically, we use `native` for `<ARCH>` to automatically target the current system, but some configurations (such as using `ifort` on AMD EPYC CPUs) required us to specify the option manually (in that case `<ARCH>` is set to `core-avx2`). All *Serial* code versions use only these default compiler flags.

`nvfortran`: For GPU parallelization, the *Original* code uses the flag `-acc=gpu` which enables the OpenACC directives. We also include the flag `-gpu=ccXY, cudaX.Y` to specify the specific GPU run time and hardware capabilities (similar to `-march` for CPUs). The `ccXY` indicates a device with compute capabilities of `X.Y`, while `cudaX.Y` tells the compiler to use the `X.Y` version of

the CUDA library. To check if/how the compiler parallelized the loops, we set
`-Minfo=accel`, which outputs parallelization information.

For the *New* code (containing DC loops), we add two new flags. The first
is `-stdpar=gpu`, which enables DC loops to be parallelized and offloaded to the
GPU[15]. The other is `-Minfo=stdpar` which outputs the compiler's parallelization
messages (similar to `-Minfo=accel`). When using `-stdpar=gpu`, unified managed
memory is automatically enabled, making all allocatable arrays unified arrays.
This means the runtime is responsible for correct and efficient CPU-GPU data
transfers during the run, and any OpenACC data movement directives on such
arrays are essentially no-ops. Static arrays are not made into unified arrays, so
manual GPU data movement is still needed for good performance (note that
`diffuse` does not make use of any static arrays). If one wants to continue to
manage the GPU data manually (using OpenACC or OpenMP data movement
directives), the option `-gpu=nomanaged` can be used.

For the *Experimental* code, since there are no directives, we simply use the
standard parallelism option of `-stdpar=gpu -gpu=ccXY,cudaX.Y`, and rely on
the compiler to automatically detect the reductions and implement them cor-
rectly, as well as manage the GPU memory using unified memory.

For CPU parallelization, the *Original* code has two implementations. One is
to use OpenMP with the `-mp` flag, and the other is to use OpenACC with the
`-acc=multicore` flag. Even though the OpenMP compilation produces slightly
better performance (as was shown in Sect. 2.4), we only use the OpenACC multi-
core option. This is because `nvfortran` currently activates OpenACC when using
`-stdpar`, so we cannot use both `-stdpar` for DC and OpenMP (as would be
needed in the *New* code) since OpenMP and OpenACC are not written to
work together (and in the *New* code case, causes a compiler error). We note
that when using OpenACC for multi-core CPU, the number of threads is con-
trolled through the runtime variable `ACC_NUM_CORES=<N>`, rather than OpenMP's
`OMP_NUM_THREADS=<N>`.

For the *Experimental* code, since there are no directives, we simply use the
standard parallelism option of `-stdpar=multicore`, relying on the compiler to
automatically detect the reductions and implement them correctly.

`gfortran`: For GPU parallelization, the *Original* code uses the flag
`-fopenacc`, which enables OpenACC directives. In addition to this flag,
the intended offload GPU must be specified. For NVIDIA GPUs, the flag
`-foffload=nvptx-none` is used (targeting specific compute capabilities is not
currently implemented). We also use the flag `-fopenacc-dim=<DIM>` to specify
the parallel topology for the offload kernels. `<DIM>` is set to three colon-separated
values that map to 'gang', 'worker' and, 'vector' sizes. Since OpenACC supports
acceleration for multiple GPU vendors, the default values for the topology may
not be optimal. Although this level of optimization is outside the scope of this
paper, we observed that the `nvfortran` compiler was using a vector length of

---

[15] For `nvfortran 21.7`, it appears that setting the `-stdpar=gpu` flag implicitly sets the
`-acc=gpu` option as well. This is an important consideration if one has OpenACC
directives that should be ignored when using `-stdpar`.

128 when compiling most OpenACC loops, so as a simple optimization, we use `-fopenacc-dim=::128` for our tests. The *New* code is not supported on the GPU with `gfortran` at this time. This is because there is no current support for DC GPU offloading.

On the CPU, the *Original* code uses `-fopenmp`, which as above, activates OpenMP directives for multi-core CPU parallelism. `gfortran` does not support direct parallelism on DC loops. Therefore, for the *New* code, we must use `gfortran`'s auto parallelization feature using the `-ftree-parallelize-loops=<N>` flag, where `<N>` is the number of threads to run on. This auto parallelization analyzes both `do` and DC loops and determines if they can be parallelized and if so, implements the parallelism. Therefore, it can be used in the case of the *New* code, as well as the *Experimental* code. Since the compiler is auto-analyzing the loops, it may detect the DC reduction loops and parallelize them correctly.

`ifort`: Since `ifort` does not currently support GPU-offloading with DC or OpenACC, we only test it with CPU parallelism to ensure switching from directives to DC does not lose our CPU parallel capabilities when using `ifort`. For all code versions, we add the flags `-fp-model precise` and `-heap-arrays` as those are standard flags we use for runs of `diffuse` to ensure robustness and precision, but they are not related to parallelization. For the *Original* code, we use the flag `-fopenmp` in order to enable OpenMP directives to produce parallel code for multicore CPUs. For the *New* code, we use the same `-fopenmp` flag as the *Original* code, as it is also used to enable automatic parallelization of DC loops. The *Experimental* code also uses the same `-fopenmp` flag. However, as the documentation states that DC reduction loops are not supported, we do not expect `ifort` to parallelize them, and rather run them in serial (although as will be shown, the current compiler version parallelizes the loops anyways, resulting in incorrect results).

4   Results

Here we show timing results for all chosen compilers, code versions, and hardware (where supported). Key questions we address are: 1) do the compilers that support GPU-acceleration with directives also support it using DC? 2) does replacing directives with DC lose CPU multicore parallelism? (i.e. do the compilers support DC for CPU multicore?) 3) for compiler-hardware combinations that support parallelizing DC, how does the performance compare to the baseline directive-based code? We first report results for the *New* code compared to the *Original* code for each compiler and hardware type, and afterwards discuss results for the *Experimental* code.

For each configuration, we run the test case of Sect. 2.2, and use the Linux program `time` to record three times: `real`, `user`, and `system`. The `real` time is the wall clock time the code took to run. The `user` time is the sum of all thread times, or how much total CPU computation time was spent. Using multiple threads should result in a lower `real` time, but a (much) higher `user` time. The `system` time is the operating system overhead, which can include CPU-GPU data transfer, as well as other overheads. All reported timings are averaged over 10 runs.

4.1   Results Using `nvfortran`

The results for the *Original* and *New* code run on the GPU with `nvfortran` are shown in Table 5. The time difference between the *Original* code and the *New* code is less than 2%, and the standard deviation over the 10 runs is around $\pm 0.1s$ for both. The slight increase in run time for the *New* code is possibly due to its use of unified memory, which can be less efficient than manually managing the GPU-CPU memory as is done through OpenACC data directives in the *Original* code. This result does not achieve a full replacement of directives with DC since not all directives were replaced in the *New* code. However, the vast majority of them were, with only a few remaining directives on the reduction loops, showing great progress in replacing directives.

Table 5. GPU timing results with `nvfortran`. Both runs used the additional compiler flag `-gpu=cc80,cuda11.4`

Code	Compiler flags	real (s)	user (s)	system (s)
Original	`-acc=gpu`	35.07	34.46	0.59
New	`-acc=gpu -stdpar=gpu`	35.67	35.01	0.54

To ensure that we did not loose CPU multicore parallelism, we show the CPU results in Table 6. We see that, like in the GPU case, replacing directives with DC yields similar runtimes to the original code. Here, the *New* code with DC runs around 3% faster than the *Original* code, but both are within the standard deviation ($\pm 15s$) of the 10 runs. Therefore, there is no loss of CPU portability when using DC with `nvfortran` for our mini-app.

Table 6. CPU timing results with `nvfortran`

Code	Compiler flags	real (s)	user (s)	system (s)
Serial		1284.59	1272.45	0.22
Original	`-acc=multicore`	224.18	26214.96	1965.06
New	`-acc=multicore` `-stdpar=multicore`	219.57	25638.37	1889.20

4.2   Results Using `gfortran`

The result for the *Original* code run on the GPU with `gfortran` is shown in Table 5. Unlike `nvfortran`, `gfortran` does not support GPU-acceleration using DC, nor

Table 7. GPU timing results with gfortran.

Code	Compiler flags	real (s)	user (s)	system (s)
Original	-fopenacc -foffload=nvptx-none -fopenacc-dim=::128	49.52	48.90	0.54
New	No Support	-	-	-

is there auto parallelization support for GPU offloading. Therefore, replacing the directives with DC currently breaks support for GPU-acceleration with gfortran. For NVIDIA GPUs, this is not a prohibitive limitation since the nvfortran compiler is freely available. However, for other accelerators (namely AMD GPUs), this loss of support may rule out using DC at this time.

Unlike for GPU-acceleration, CPU multi-core parallelism with gfortran is not lost with DC, even though there is no direct support for DC parallelization. In Table 8, we show the CPU timing results of the *Original* and *New* codes. We see that the performance difference between the codes is ∼10%, within the standard deviations of the 10 runs (±13s for the *Original* and ±18s for the *New* code). The *New* code is also able to be parallelized because gfortran treats DC loops as regular do loops, which are parallelized using the auto parallelization feature. However, the loops in diffuse are fairly simple. In other codes, the auto parallelization may not be able to handle more complex loops, that could otherwise be parallelized using directives. Therefore, the result here should be viewed with caution. Note also that the auto parallelization feature works the same on our code with regular do loops as it does with DC loops (i.e. it would even parallelize the *Serial* code version).

Table 8. CPU timing results with gfortran

Code	Compiler flags	real (s)	user (s)	system (s)
Serial		1308.75	1296.74	0.16
Original	-fopenmp	191.90	24117.72	8.02
New	-fopenmp -ftree-parallelize-loops=128	212.64	26588.59	8.65

The thread control for the *New* code is unique. Since the OpenMP directives are still on the reduction loops, while the remaining loops use DC with no directives, the number of CPU threads used for the reductions is set by the standard OMP_NUM_THREADS environment variable, while that used by the DC loops is controlled by the compiler flag value -ftree-parallelize-loops=<N>. This complicates the thread control, and also removes run-time thread control.

4.3   Results Using `ifort`

As mentioned in the introduction, the Intel OpenAPI Toolkit does not currently support GPU-acceleration using DC loops, but there are plans for support in the future. Therefore, here we focus on DC compatibility with multi-core CPU parallelism. In Table 9, we show the timing results for the *Original* and *New* codes. We see that replacing directives with DC still allows for multi-core CPU parallelism, and surprisingly exhibits a nearly 10% improvement in performance. The standard deviation of the 10 runs was roughly $\pm 8s$, so this performance increase is significant. It may be attributed to more efficient optimizations being available to the compiler when using DC compared to OpenMP directives.

Table 9. CPU timing results with `ifort`.

Code	Compiler flags	real (s)	user (s)	system (s)
Serial		1318.60	1306.27	0.18
Original	`-fopenmp`	194.86	24213.11	320.53
New	`-fopenmp`	178.29	21888.21	280.65

Since the implementation of DC parallelism is connected to OpenMP (as indicated by the use of the `-fopenmp` flag), the number of threads remains controlled by the standard environment variable `OMP_NUM_THREADS` (or optionally at compile time with `-par-num-threads=<N>` which overrides `OMP_NUM_THREADS`).

4.4   Experimental Results

As mentioned in Sect. 3.2, the current Fortran standard does not have a way to indicate to the compiler that a DC loop requires reduction or atomic operations. However some compilers have implemented code analysis methods to automatically detect and implement such operations. Therefore, the *Experimental* code, which represents the ideal scenario of replacing all directives with DC loops, may work with some compilers.

Using `nvfortran`, we found that the code parallelized and ran correctly on both the GPU and CPU. It appears `nvfortran` detects the reductions and implements them correctly for our code. The run times are shown in Table 10. They are nearly

Table 10. GPU and CPU timing results for the *Experimental* code with `nvfortran`.

Code	CPU/GPU	Compiler flag	real (s)	user (s)	system (s)
Experimental	GPU	`-stdpar=gpu` `-gpu=cc80,cuda11.4`	35.63	35.01	0.53
Experimental	CPU	`-stdpar=multicore`	219.21	25654.35	1906.40

identical to those of the *New* code shown in Tables 5 and 6 which is expected since only a few small loops used directives for reductions in the *New* code. This means that for `diffuse`, we can use DC to fully eliminate all directives and not lose any CPU or GPU performance with `nvfortran`. We reiterate that more complicated codes may not yet work with zero directives for a variety of reasons including not detecting complicated reduction or atomic operations, not being compatible with in-lined function calls, and, for GPU-acceleration, not supporting automatic memory management with static arrays.

For `gfortran`, we only test the code on the CPU since there is no support for DC GPU-offloading. In this case, the *Experimental* code ran correctly, implying the auto-parallelization done by the compiler was able to detect the reductions and parallelize them. The run time is shown in Table 11 and is roughly 10% slower comparable to the run time of the *New* code in Table 8. However, the times are nearly within the standard deviation of the 10 runs ($\pm 18s$).

Table 11. CPU timing results for the *Experimental* code with `gfortran`.

Code	Compiler flags	real (s)	user (s)	system (s)
Experimental	`-ftree-parallelize-loops=128`	236.28	29565.01	10.08

Using `ifort`, the *Experimental* code compiled and ran, but did not give the correct results. This is because `ifort` does not support implicit reductions of DC loops, yet parallelized the loop anyways when we used the `-fopenmp` flag, Therefore, the resulting inherent race conditions produced incorrect results.

5 Discussion

In this paper, we have used a mini-app code to explore the current status of replacing `do` loops using directives with `do concurrent` (DC) loops for accelerated computing. The original code used OpenACC for GPU-acceleration when compiled with `gfortran` or `nvfortran`, and OpenMP for multi-core CPU parallelism when compiled with `gfortran`, `nvfortran`, or `ifort`. We modified the code to replace the directives with DC and used a test case to explore the resulting compatibility, portability, and performance, all with the newest available versions of the compilers.

Compatibility: We found that only `nvfortran` currently supports GPU acceleration with DC, and therefore replacing the directives removed GPU support when using `gfortran`. Since `nvfortran` is freely available, this is not an insurmountable problem when running on NVIDIA GPUs. However, `gfortran` also has AMD (and possible future Intel) GPU support, making this an important consideration. The `ifort` compiler does not currently support GPU-acceleration with DC, but as Intel has indicated plans to add this support soon, switching from OpenACC directives

to DC may increase compatibility (as `ifort` only supports OpenMP GPU offload, not OpenACC).

We also found that the current Fortran specification for DC lacks features that are needed to guarantee correct parallelization of all of our mini-app's parallelizable loops (specifically, loops with reductions). Indeed, when `ifort` attempted to parallelize our reduction loops for the CPU, it resulted in incorrect results. In contrast, the `nvfortran` compiler has implicit reduction detection of DC loops, allowing us to replace all directives with DC. The next release of the Fortran standard (202x) will include an explicit 'reduce' clause on DC, which, when implemented, should alleviate this issue.

Another compatibility concern is that we currently use OpenACC directives to manually control GPU-CPU memory management, and removing these could cause extreme loss of performance. In the case of `nvfortran`, since it automatically activates its unified memory management feature when compiling with DC GPU-acceleration, this issue is avoided. However, unified memory is limited to allocatable arrays, so static arrays may still require data management directives.

Using cutting-edge language and compiler features have a risk of breaking backward compatibility with older compilers. In this paper we used the most recent versions of the compilers we could for the best support, but on some systems this is not always available. Container frameworks like Singularity/Apptainer (as used here) can help mitigate this issue, however the frameworks are also not always available on all systems, and can sometimes be complicated to use for large scale simulations.

Portability: A key consideration in replacing directives with DC for GPU acceleration was to see if, by doing so, we still maintain CPU multi-core parallelism (that we originally used OpenMP directives to achieve). We found that `nvfortran` and `ifort` compilers directly support DC for multi-core CPU parallelism, while `ifort` requires directives on loops with reductions for correctness. With `gfortran`, while there is no direct support for DC parallelism, the loops can still be parallelized using `gfortran`'s auto parallelization feature. With this feature, even reduction DC loops are correctly recognized and parallelized. Thus, all three compilers we use are able to keep multi-core CPU parallelism when replacing directives with DC (with `ifort` still requiring some on reduction loops).

Performance: Replacing directives with DC allows much cleaner looking code and robustness due to being part of the standard language. However, this is only worth while if it also results in acceptable performance. Through our timings, we found that in both the GPU and CPU cases, the performance of the code after replacing directives with DC was comparable to that of the original directive-based code, with some configurations improving performance slightly, and in others, decreasing slightly. For GPU runs with DC, `nvfortran`'s unified memory was used, and the resulting performance was comparable to using manual OpenACC data directives. However, more complicated codes may not be as compatible with unified memory and/or may lose some performance using it.

Summary: With `nvfortran`, we were able to remove and replace all directives in our code with DC, and achieve efficient CPU and GPU parallelism. However,

this relied on specific features of nvfortran including implicitly detecting reductions and the use of unified managed memory. In order to maintain cross compiler compatibility, we can continue to use OpenACC/OpenMP directives for reductions and data movement until equivalent standard language features are written and widely supported. Even with the remaining directives, using DC has a large benefit, as the number of directives is decreased dramatically.

Can Fortran's do concurrent replace directives for accelerated computing? With nvfortran and NVIDIA GPUs, for some codes (such as ours) the answer is yes, and with no (or minimal) loss of performance. With upcoming language features and compiler implementations, more complicated codes may also eventually be parallelized without directives, and do so with support across multiple compiler and hardware vendors.

Appendix

Singularity/Apptainer containers: In order to test the latest compilers and to simplify setup of our library dependencies, we utilized Singularity/Apptainer containers. These containers allow one to run software in a containerized environment on any compatible system using only the container file.

Container setup: The containers were straight forward to setup and use for our timings. Two methods were used to create them. For nvfortran and ifort, a docker image of NVIDIA HPC SDK or Intel OneAPI HPC Toolkit was used to create a sandbox. For gfortran, a similar sandbox with Ubuntu 21.04 was created and then gfortran was installed with the apt-get command. Once the sandboxes were created, the dependent libraries were installed. A sandbox is treated like a virtual machine, allowing us to edit and install new software into the container (note this requires sudo privileges). Once all the needed software is installed, the sandbox is converted to a .sif file which can be copied and run (without sudo privileges) on any other compatible machine with Singularity/Apptainer installed, but it can no longer be edited. However, the container is able to modify files outside itself, allowing us to compile and run the test cases. For GPU-accelerated runs, a special flag is needed when running the container depending on the vendor of GPU. For NVIDIA GPUs, the flag is --nv, while for AMD GPUs, the flag is --rocm. For more details on Singularity/Apptainer containers, see Ref. [7].

Container performance tests: Using containers can sometimes cause performance overhead. To ensure that using the Singularity/Apptainer containers does not cause significant overhead in our case, we ran two test cases on both a bare metal setup and with a container with the same compiler version (in this case gfortran 10.2). Table 12 shows timings of the test run using both the *Original* and *Serial* codes described in Sect. 3.2. We see that the runs using the container perform nearly identical to those run on bare metal, allowing us to confidently use the containers for the runs in the paper.

Table 12. Timing results on a Bridges2 CPU compute node using `gfortran` 10.2 bare metal and form within a Singularity Container

Code	Run method	real (s)	user (s)	system (s)
Serial	Bare Metal	1306.10	1294.30	0.154
	Singularity	1300.43	1287.50	0.168
Original	Bare Metal	164.87	20782.32	5.935
	Singularity	165.27	20777.85	7.248

Reproducibility package: The results in this paper can be reproduced using our reproducibility package hosted publicly at Ref. [9] and on our website[16]. The package contains three Singularity/Apptainer containers (for `gfortran`, `nvfortran`, and `ifort`), as well as all code versions, compiler options, and test cases. The package requires minimal customization (only specifying hardware-specific compiler options) of the main script, which can then be used to automatically run either all, or a subset, of runs from the paper. See the documentation in the package for more details. A reference solution is also provided for validation. Note that runs using GPU-acceleration require having an NVIDIA GPU with compatible drivers installed on the system.

6   Artifact Availability Statement

Summary of the Experiments Reported

Timings were performed on the various mini-app versions on the CPU and GPU using singularity containers. These versions represented different levels of replacing OpenACC and OpenMP directives with do concurrent loops. Each version was tested with the compilers gfortran, nvfortran, and ifort. The compilers were loaded in a singularity container, and the codes were executed through these singularity containers. For each code and compiler version, 10 runs were carried out in order to get an average run time and standard deviation.

Artifact Availability

Software Artifact Availability: All author-created software artifacts are maintained in a public repository under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: All author-created data artifacts are maintained in a public repository under an OSI-approved license.

---

[16] www.predsci.com/papers/dc.

Proprietary Artifacts: No author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:

```
10.5281/zenodo.5253520
http://www.predsci.com/papers/dc
```

References

1. Balarac, G., et al.: AVBP and YALES2 portability, tuning and scalability on AMD EPYC 7002 Rome processors (2020)
2. Caplan, R.M., Downs, C., Linker, J.: Preparing photospheric magnetic field measurements for use in coronal and heliospheric models. In: AGU Fall Meeting Abstracts, vol. 2019, pp. SH43E–3389 December (2019)
3. Caplan, R.M., Mikić, Z., Linker, J.A., Lionello, R.: Advancing parabolic operators in thermodynamic MHD models: explicit super time-stepping versus implicit schemes with krylov solvers. J. Phys. Conf. Series **837**, 012016 (2017). https://doi.org/10.1088/1742-6596/837/1/012016
4. Caplan, R.M., Downs, C., Linker, J.A., Mikic, Z.: Variations in finite-difference potential fields. Astrophys. J. **915**(1), 44 (2021). https://doi.org/10.3847/1538-4357/abfd2f
5. Chandrasekaran, S., Juckeland, G.: OpenACC for Programmers: Concepts and Strategies. Addison-Wesley Professional (2017)
6. David Olsen, Graham Lopez, B.A.L.: Accelerating standard C++ with GPUs using stdpar (2021).https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/
7. Kurtzer, G.M., Sochat, V., Bauer, M.W.: Singularity: scientific containers for mobility of compute. PLOS ONE **12**(5), e0177459 (2017). https://doi.org/10.1371/journal.pone.0177459
8. Meyer, C.D., Balsara, D.S., Aslam, T.D.: A stabilized Runge-Kutta-Legendre method for explicit super-time-stepping of parabolic and mixed equations. J. Comput. Phys. **257**, 594–626 (2014). https://doi.org/10.1016/j.jcp.2013.08.021
9. Mikic, Z., Caplan, R.M., Linker, J.A., Stulajter, M.: Reproducibility package for running the DIFFUSE test cases from "Can Fortran's 'do concurrent' replace directives for accelerated computing" (2021). https://doi.org/10.5281/zenodo.5253520
10. Ozen, G., Lopez, G.: Accelerating Fortran do concurrent with GPUs and the NVIDIA HPC SDK (2020). https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/
11. Van der Pas, R., Stotzer, E., Terboven, C.: Using OpenMP The Next Step: Affinity. Tasking, and SIMD. MIT press, Accelerators (2017)
12. Towns, J., Cockerill, T., Dahan, M., Foster, I., Gaither, K., Grimshaw, A., Hazlewood, V., Lathrop, S., Lifka, D., Peterson, G.D., Roskies, R., Scott, J.R., Wilkins-Diehr, N.: XSEDE: accelerating scientific discovery. Comput. Sci. Eng. **16**(5), 62–74 (2014). https://doi.org/10.1109/mcse.2014.80
13. Xinman Tian, Kari Qi, M.L.: Practical examples of OpenMP offload to GPUs (2021).https://techdecoded.intel.io/essentials/3-quick-practical-examples-of-openmp-offload-to-gpus/