Framework for Extensible, Asynchronous Task Scheduling (FEATS) in Fortran

Brad Richardson[1(✉)], Damian Rouson[1,2], Harris Snyder[1], and Robert Singleterry[3]

[1] Archaeologic Inc., Oakland, CA, USA
{brad,damian,harris}@archaeologic.codes
[2] Lawrence Berkeley National Laboratory, Berkeley, CA, USA
rouson@lbl.gov
[3] NASA Langley Research Center, Hampton, VA, USA
robert.c.singleterry@nasa.gov
https://archaeologic.codes, https://www.lbl.gov,
https://www.nasa.gov/langley

Abstract. Most parallel scientific programs contain compiler directives (pragmas) such as those from OpenMP [1], explicit calls to runtime library procedures such as those implementing the Message Passing Interface (MPI) [2], or compiler-specific language extensions such as those provided by CUDA [3]. By contrast, the recent Fortran standards empower developers to express parallel algorithms without directly referencing lower-level parallel programming models [4,5]. Fortran's parallel features place the language within the Partitioned Global Address Space (PGAS) class of programming models. When writing programs that exploit data-parallelism, application developers often find it straightforward to develop custom parallel algorithms. Problems involving complex, heterogeneous, staged calculations, however, pose much greater challenges. Such applications require careful coordination of tasks in a manner that respects dependencies prescribed by a directed acyclic graph. When rolling one's own solution proves difficult, extending a customizable framework becomes attractive. The paper presents the design, implementation, and use of the Framework for Extensible Asynchronous Task Scheduling (FEATS), which we believe to be the first task-scheduling tool written in modern Fortran. We describe the benefits and compromises associated with choosing Fortran as the implementation language, and we propose ways in which future Fortran standards can best support the use case in this paper.

Keywords: Modern Fortran · Task Scheduling · Framework · coarray

1 Introduction

Modern computing hardware has evolved to offer a variety of opportunities to exploit parallelism for high performance – including multicore processors with

P. Diehl et al. (Eds.): WAMTA 2023, LNCS 13861, pp. 39–51, 2023.
https://doi.org/10.1007/978-3-031-32316-4_4

vector units, superscalar pipelines, and embedded or off-chip graphics processing units. Exploiting the abundance of opportunities for parallel execution requires searching for a variety of forms of parallelism. Chief among the common parallel programming patterns are data parallelism and task parallelism [6]. Parallel programming languages have evolved native features that support data parallelism. In Fortran 2018, for example, such features include giving the programmer the ability to define teams (hierarchical sets) of images (processes) that execute asynchronously with each image having one-sided access to other team members' local portions of "coarray" distributed data structures [4]. These features have now seen use in production codes running at scale for simulating systems ranging from weather [7] and climate [8] to plasma fusion [9].

By contrast, task parallelism generally proves much more challenging for application developers to exploit without deep prior experience in parallel programming. Although data parallelism maps straightforwardly onto a bulk synchronous programming model in which periods of computation are interspersed with periods of communication followed by barrier synchronization, efficient execution of independent tasks generally requires asynchronous execution with more loose forms of coordination such as semaphores. To wit, it takes roughly 15 source lines of code to implement a bulk synchronous "Hello, world!" program using Fortran's barrier synchronization mechanism, the `sync all` statement; whereas it takes more than three times as many lines to write a similar, asynchronous program taking advantage of Fortran's `event_type` derived type, the language's mechanism supporting semaphores [10].

A central challenge in writing asynchronous code to coordinate tasks centers around task parallelism's more irregular execution and communication patterns. Whereas partial differential equation solvers running in a data parallel manner typically involve a predictable set of halo data exchanges between grid partitions at every time step, task parallelism generally enjoys no such regular communication pattern. Programmers generally represent task ordering requirements in a Directed Acyclic Graph (DAG) of task dependencies [11]. Tasks can execute in any order that respects the DAG. Moreover, the DAG can change considerably from one problem to the next and even from one execution to the next. For example, a DAG describing the steps for building a software package will vary over the life of the software as internal and external dependencies change.

Writing code to handle the level of flexibility needed efficiently is daunting for most application developers, which makes the use of a task-scheduling framework attractive. Fortran programmers face the additional challenge that the task scheduling frameworks of which the authors are aware are written in other programming languages such as C++ [12] and UPC++ [13]or target specific domains such as linear algebra [14]. FEATS aims to support standard Fortran 2018 with a standard Fortran 2018 framework and is unique in these aspects.

Rumors of Fortran's demise are greatly exaggerated. Despite longstanding calls for Fortran's retirement [15] and descriptions of Fortran as an "infantile disorder," [16] the world's first widely used high-level programming language continues to see important and significant use. Fortran is arguably enjoying a renaissance characterized by a growing list of new compiler projects over the

past several years and a burgeoning community of developers at all career stages writing new libraries [17], including some in very non-traditional areas such as package management [18]. The National Energy Research Scientific Computing Center (NERSC) used system monitoring of runtime library usage to determine that approximately 70% of projects use Fortran [19] and found that the vast majority of projects use MPI.

In MPI, the most advanced way to achieve the aforementioned requirements of loosely coordinated, high levels of asynchronous execution required for efficient task scheduling involves the use of the one-sided `MPI_Put` and `MPI_Get` functions introduced in MPI-3. In the authors' experience, however, the overwhelming majority of parallel MPI applications use MPI's older two-sided communication features, such as the non-blocking `MPI_ISend` and `MPI_IRecv` functions partly due to the challenges of writing one-sided MPI. Our choice to write and support Fortran's native coarray communication mechanism enables us to take advantage of the one-sided MPI built into some compiler's parallel runtime libraries, e.g., in the OpenCoarrays [20] runtime used by `gfortran`, or whatever communication substrate a given compiler offeror chooses to best suit particular hardware. Moreover, this choice implies that switching from one communication substrate to another might require no more than switching compilers or even swapping compiler flags and ultimately empowers scientists and engineers to focus more on the application's science and engineering and less on the computer science.

2 Implementation

FEATS itself consists of eight Fortran modules. Before they can be described, there is one key upstream dependency which must be noted: `dag`, a separate library for manipulating directed acyclic graphs in Fortran. Using `dag`, directed acyclic graphs can be assembled directly in Fortran code, or as a JSON (JavaScript Object Notation) file which is read at run time. FEATS leverages the `dag` library to store the graph of tasks to be executed.

FEATS is designed around the use of Fortran coarrays to provide distributed multiprocessing and data exchange between application images. FEATS automatically assigns the first image to be the "scheduler" image, responsible for tracking what work has been completed and which tasks can be executed next based on the task dependency graph, and assigning work to the other ("compute") images. The `image_m` module provides an `image_t` derived type, which encapsulates the data required for the operation of an image and exposes a single "run" procedure. That run function is given an `application_t` object (provided in the module `application_m`), which stores a list of task objects (described below) and a `dag` graph, which describes the dependencies between tasks.

Tasks in FEATS are represented as objects. FEATS provides an abstract derived type `task_t`, which the user should extend in their own derived type definition, and provide the necessary "execute" function required to complete the task. This design is convenient for the user, but a side-effect is that the tasks will be of different types (granted, with a common base type). Since Fortran does

not allow an array to be created where the elements of the array have different types, it was necessary to create a wrapper type, `task_item_t`, which stores a `class(task_t)` as an allocatable member. With this wrapper type, an array of `task_item_t` values can be created and stored. While an implementation detail, in general, a user will not have to interact with `task_item_t` in order to simply *use* FEATS.

Tasks have inputs and outputs, so there must be some mechanism by which to transmit those inputs and outputs between images. This transmission is done using coarrays, though it should be noted that all image control and coarray code is internal to the FEATS library, meaning that the user need not directly deal with any details related to parallel programming, or even understand coarrays. The "execute" function of each task type can accept and return `payload_t` objects, which encode task inputs and outputs. Different tasks will of course have different input and output types based on their purpose, which brings up another difficulty of implementing FEATS as a library. Since the library code cannot know the details of different tasks' input and output types, it must represent these payloads in some generic way so that it can be transmitted between images. FEATS solves the problem by storing payloads as an array of integers (just a string of bytes in memory), and the user must use the Fortran `transfer` statement to serialize their data into and out of payloads. This serialization does come with some caveats; the user needs to ensure that the types they use as payloads can be serialized and deserialized safely (for example, a simple derived type with inline elements will work correctly, whereas one with pointers and allocatable components likely will not). Although arguably an aesthetically "inelegant" approach, the authors see it as an acceptable engineering tradeoff in the interest of generality.

The `mailbox_m` module contains the actual payload coarrays used for data exchange, and the `final_task_m` module contains a task type that serves as a signal to the compute images that all work has been completed. Both of these modules are implementation details, and the user never needs to interact with them directly.

The final module constituting FEATS is `feats_result_map_m`. Tasks, particularly ones at the graph's terminal nodes, may have outputs which the user wants to access after the whole graph has been processed. The aim of the `feats_result_map_m` module is to offer a derived type that tracks which image has the results from terminal nodes in the graph. As of this writing, implementation of the type has not yet been completed. Implementation of such a type should be fairly straightforward, and we plan to add it.

3 Advantages, Disadvantages, and Examples

This section discusses how the features of Fortran enable/support the development of FEATS, and aspects of the language that currently serve as impediments to the desired features of the framework.

3.1 Advantages

There are several features of the modern Fortran language that make it a natural fit for implementing a task scheduling framework. Several aspects have featured prominently in the implementation, but in this section we will discuss what makes them beneficial for implementing a task scheduling framework.

Coarrays and Events. The fundamental problem of task scheduling requires methods of communicating data between tasks, and coordinating the execution of those tasks to enforce prerequisite tasks are completed before subsequent tasks begin. The coarray feature of Fortran provides a simple and effective method of performing one-sided communication between the scheduler and executor images to facilitate data transfer between tasks. While other languages and libraries have methods of communicating data between processes, they often require two-sided operations (i.e. both processes must participate in the communication), require calls to external library procedures, or require significant expertise to use correctly. Having the communication facilities as a native feature of the language simplifies the syntax and implementation complexities and reduces the number of external dependencies.

Although other language and library communication methods are generally sufficient for implementing coordination mechanisms, doing so manually requires a high level of expertise and adds complexity to the implementation. Having a native feature of the language explicitly designed for the purposes of coordination, namely event types, again simplifies the syntax and implementation complexities and reduces the number of external dependencies.

Teams. Although there may be task scheduling algorithms that do not require a reserved process to act as a scheduler, these algorithms generally come at the cost of increased overhead in terms of coordination and complexity of implementation. However, having a dedicated scheduler can introduce a communication and coordination bottleneck in cases of large tasks graphs being executed by large numbers of processes. While we have not yet implemented it, the teams feature of Fortran allows for a simple and natural partitioning of processes such that multiple schedulers can coordinate with segments of executor images operating on partitions of the task graph.

Polymorphism. Although it may be possible to implement a task-scheduling framework without polymorphism, it would require implementation of a predetermined set of possible task interfaces, which would likely be limiting for potential users. By making use of abstract type definitions and type-extension, and defining a generic interface for a task, the procedure of defining a task and including it in a graph becomes a natural process for users, with help from the compiler in enforcing that they have done so properly. The process of defining new tasks involves creating a new derived type which extends from the framework's `task_t` type and providing an implementation for the run procedure. A task can then be created by instantiating an object of this new type, to be included in the dependency graph.

Fortran's History. Fortran's long history of use in scientific computing means that there are likely a large number of applications that could benefit from a Fortran-specific task scheduling framework. We have already identified a potential target application in NASA's OLTARIS [21], space radiation shielding software. Other prime target applications are those which perform a series of different, but long running calculations, or those which perform parallel calculations (or easily could), but which experience load balancing issues.

3.2 Disadvantages

There are some ways in which the Fortran language lacks some important features that would allow for an even better implementation. We will discuss these shortcomings and the ways in which the language could be improved to address them, or how they can be worked around.

Data Communication. The lack of ability to utilize polymorphism in coarrays means that communication of task input and output data cannot be done as seamlessly as users would like. In order to communicate the inputs and outputs between tasks, users are forced to manually serialize and deserialize the data into a pre-defined format for transfer between processes. This means it will also be difficult for users to make use of polymorphism in their calculations, as deserialization of polymorphic objects can be done only with a predefined set of possible result types. Further, the lack of ability to communicate polymorphic objects via coarrays means that each image must have a complete copy of the dependency graph and its tasks, because the tasks themselves cannot be communicated to the executing images later. This represents a moderate inefficiency in data storage and in initial execution for each image to compute/construct the dependency graph. A strategic relaxation of a single constraint in the Fortran standard is all that would be required to enable the use of polymorphism in the data communication.

Task Detection, Fusion or Splitting. Because Fortran lacks any features for introspection or reflection, it is not possible for the framework to automatically detect tasks, fuse small tasks together, or split large tasks apart. All task definition must be performed manually by the user, with no help from the framework. It would be possible to allow users to manually provide information about task and data sizes to encourage certain sequences of tasks to be executed on one image, but would likely be difficult and error prone. Future work could involve exploring avenues for annotating tasks to help the scheduler more efficiently assign tasks to images.

Task Independence. Task independence is a problem for all task based applications, but Fortran provides few avenues for mitigating or catching possible mistakes. Any data dependencies between tasks not stated explicitly in the dependency graph and communicated as arguments to the task or its output

allow for the possibility of data races. In other words, all tasks must be pure functions with all dependencies defined. Many existing Fortran applications were not written in this style, and may require extensive work to refactor to a form in which they could take advantage of a task scheduling framework. It is the opinion of the authors that most applications could benefit from such refactoring to enable parallel execution regardless of the desire to use this framework, but understand that the costs involved do not always make this refactoring feasible. Users could make these dependencies explicit without using the framework to transmit the data, but it may be beneficial to develop tools to help users identify these "hidden" dependencies.

Lagging Compiler Support. While the features necessary for developing this framework have been defined by the language standard since 2018, compilers have been slow to implement them, and support is still buggy and lacking. For example, we were able to work around a bug in gfortran/OpenCoarrays regarding access of allocatable components of derived types in a corray on remote images by defining the payload size to be static for the purposes of demonstrating the examples shown below. The other compilers with support for the parallel features have other bugs which have thus far not allowed us to compile and execute the examples with them. Specifically Intel's ifort/ifx has a bug which reports finding duplicate symbols when compiling one of our dependencies. The Numerical Algorithm Group's (NAG) nagfor has a few bugs related to declaring coarrays in submodules. Cray/HPE's Fortran compiler reports mismatches between the interface declared for a procedure in a module and that specified in the submodule, despite the interface not being redeclared in the submodule. We have reported these bugs to those compilers, and are awaiting their resolution to try our framework with them.

3.3 Examples

The examples described in this section can be found in the FEATS repository at https://github.com/sourceryinstitute/feats. In order to give the reader a sense of the compiler landscape, we present one example that is blocked by bugs in current compilers and one example that works correctly with at least one currently available compiler.

A Quadratic Root Finder. The typical algorithm/equation for finding the roots of a quadratic equation can be defined as tasks and FEATS can then be used to perform the calculations. The use of such a simple example can be beneficial for demonstrating the use of the framework. Given a quadratic equation of the form:

$$a * x^2 + b * x + c = 0 \tag{1}$$

then the equation to determine the values of x which satisfy the equation (the roots), is:

$$\frac{-b \pm \sqrt{b^2 - 4 * a * c}}{2 * a} \tag{2}$$

The diagram in Fig. 1 illustrates how this equation can be broken into separate steps and shows the dependencies between them.

The equivalent FEATS application can be constructed as follows, assuming the tasks have been appropriately defined. We also note that the dag library used (and thus the solver object) is capable of producing (and was used to produce nearly exactly) the graphviz source code used to generate the image in Fig. 1.

```
solver = dag_t( &
    [ vertex_t([integer::], "a") &
    , vertex_t([integer::], "b") &
    , vertex_t([integer::], "c") &
    , vertex_t([2], "#**2") &
    , vertex_t([1, 3], "4*#*#") &
    , vertex_t([4, 5], "sqrt(# - #)) &
    , vertex_t([2, 6], "-# +- #") &
    , vertex_t([1], "2*#") &
    , vertex_t([8, 7], "# / #") &
    , vertex_t([9], "print roots") &
    ])
tasks = &
    [ task_item_t(a_t(2.0)) &
    , task_item_t(b_t(-5.0)) &
    , task_item_t(c_t(1.0)) &
    , task_item_t(b_squared_t()) &
    , task_item_t(four_ac_t()) &
    , task_item_t(square_root_t()) &
    , task_item_t(minus_b_pm_square_root_t()) &
    , task_item_t(two_a_t()) &
    , task_item_t(division_t()) &
    , task_item_t(printer_t()) &
    ]
application = application_t(solver, tasks)
```

This example produces output like the following, with a slightly different order of execution being possible each time except that an operation is never performed prior to the results of the operations on which it depends.

```
c =      1.00000000
b =     -5.00000000
a =      2.00000000
2*a =      4.00000000
b**2 =     -5.00000000
4*a*c =      8.00000000
sqrt(b**2 - 4*a*c) =      4.12310553       -4.12310553
-b +- sqrt(b**2 - 4*a*c) =      9.12310600       0.876894474
(-b +- sqrt(b**2 - 4*a*c)) / (2*a) =      2.28077650       0.219223619
The roots are      2.28077650       0.219223619
```

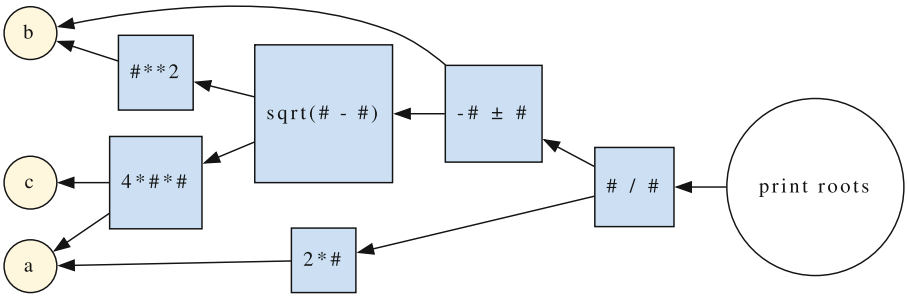

Fig. 1. Graphical representation of the computational tasks involved in calculating the roots of a quadratic equation.

Compiling FEATS. Compiling software projects is a common example of an application involving tasks. By defining the dependencies between files, and defining their compilation as a task, it becomes possible to use FEATS to compile itself. The FEATS source file dependencies are described by the image in Fig. 2, and the FEATS application can be constructed as follows.

```
feats = dag_t(&
  [ vertex_t([integer::], name_string(assert_m)) &
  , vertex_t([integer::], name_string(dag_m)) &
  , vertex_t( &
    [dag_m, task_item_m], name_string(application_m)) &
  , vertex_t( &
    [assert_m, application_m], &
    name_string(application_s)) &
  , vertex_t( &
    [integer::], name_string(feats_result_map_m)) &
  , vertex_t( &
    [payload_m, task_m], name_string(final_task_m)) &
  , vertex_t([final_task_m], name_string(final_task_s)) &
  , vertex_t( &
    [application_m, feats_result_map_m, payload_m], &
    name_string(image_m)) &
  , vertex_t( &
    [dag_m, final_task_m, image_m, &
    mailbox_m, task_item_m], &
    name_string(image_s)) &
  , vertex_t([payload_m], name_string(mailbox_m)) &
  , vertex_t([integer::], name_string(payload_m)) &
  , vertex_t([payload_m], name_string(payload_s)) &
  , vertex_t( &
    [payload_m, task_m], name_string(task_item_m)) &
  , vertex_t([task_item_m], name_string(task_item_s)) &
```

```
  , vertex_t([payload_m], name_string(task_m)) &
  , vertex_t([task_m], name_string(task_s)) &
  ])
tasks = [(task_item_t(compile_task_t(name_string(i))), &
  i = 1, size(names))]
application = application_t(feats, tasks)
```

This example produces output like the following, with a slightly different order of execution being possible each time except that a file is never started compiling prior to a file it depends on first completing its compilation, and with a possibly different image executing each task.

```
Compiling: dag_m on image number: 3
Compiling: assert_m on image number: 4
Compiling: feats_result_map_m on image number: 2
Finished Compiling: assert_m
Compiling: payload_m on image number: 4
Finished Compiling: dag_m
Finished Compiling: feats_result_map_m
Finished Compiling: payload_m
Compiling: mailbox_m on image number: 4
Compiling: task_m on image number: 2
Compiling: payload_s on image number: 3
Finished Compiling: mailbox_m
Finished Compiling: task_m
Compiling: task_item_m on image number: 2
Finished Compiling: payload_s
Compiling: task_s on image number: 3
Compiling: final_task_m on image number: 4
Finished Compiling: final_task_m
Compiling: final_task_s on image number: 4
Finished Compiling: task_item_m
Compiling: application_m on image number: 2
Finished Compiling: application_m
Compiling: application_s on image number: 2
Finished Compiling: task_s
Compiling: image_m on image number: 3
Finished Compiling: final_task_s
Compiling: task_item_s on image number: 4
Finished Compiling: task_item_s
Finished Compiling: application_s
Finished Compiling: image_m
Compiling: image_s on image number: 4
Finished Compiling: image_s
```

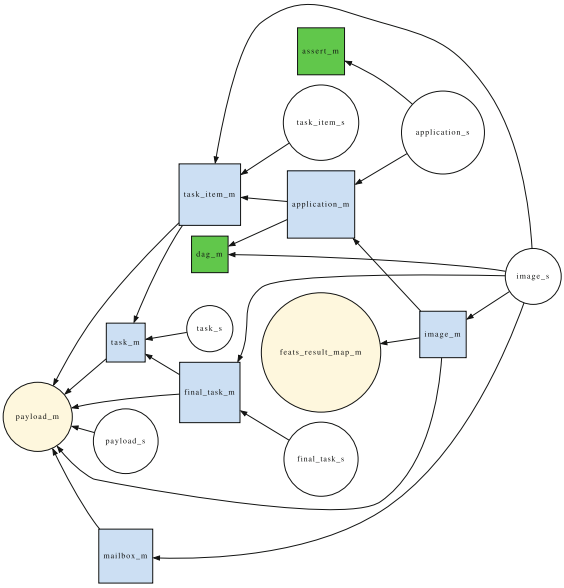

Fig. 2. Graphical representation of the tasks involved in compiling the FEATS library.

4 Conclusion

We believe the existing Fortran applications, and the Fortran ecosystem generally, would greatly benefit from a native tasking framework. The prototype implementation of FEATS has successfully demonstrated that implementing a task scheduling framework in Fortran is feasible. Working around limitations of the language and the bugs in various compilers' coarray feature implementation has proven a challenging but not impassible barrier. With this demonstration of a working prototype implementation, we have taken a significant first step towards providing such a capability to Fortran users.

We look forward to working on several unresolved issues in FEATS. The first order of business will be to implement the `result_map_t` type to allow for further processing of results after completed execution of a task graph. Also, we will submit and follow up on bug reports to the writers of the compilers that we have attempted to use for executing the examples presented. Further, we will begin to explore the performance characteristics of the framework as we use the framework to execute larger task graphs on machines with larger numbers of processors.

Longer term work planned will involve collaborating with the Fortran standard committee to add capabilities to the language that will enable FEATS behaviors such as communication of polymorphic objects between images using coarrays. We have identified a targeted relaxation of a specific constraint in the standard to allow for the needed functionality. We will also explore graph parti-

tioning algorithms and the use of the Fortran 2018 teams feature to potentially improve the ability of the framework to scale to large problems and systems. We also hope to find potential users of the framework and help them to integrate it into their applications. Possible initial target applications include parallel builds with the Fortran package manager [18] and works-stealing with the Intermediate Complexity Atmospheric Research model [8].

References

1. Miguel, H.: Parallel programming in Fortran 95 using OpenMP. Technique report, Universidad Politecnica De Madrid (2002)
2. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 4.0 (2021). https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
3. Reutsch, G., Fatica, M.: CUDA Fortran for scientists and engineers: best pracices for efficient CUDA Fortran programming. Elsevier (2013)
4. Numrich, R.: Parallel Programming with Co-Arrays. Chapman and Hall/CRC, Boca Raton (2018)
5. Curcic, M.: Modern Fortran: Building Efficient Parallel Applications. Manning Publications (2021)
6. Massingill, B., Sanders, B., Mattson, T.G.: Patterns for Parallel Programming. Pearson Education, United Kingdom (2004)
7. Mozdzynski, G., Hamrud, M., Wedi, N.: A partitioned global address space implementation of the European centre for medium range weather forecasts integrated forecasting system. Int. J. High Perform. Comput. Appl. **29**(3), 261–273 (2015)
8. Gutmann, E., Barstad, I., Clark, M., Arnold, J., Rasmussen, R.: The intermediate complexity atmospheric research model (ICAR). J. Hydrometeorol. **17**(3), 957–973 (2016)
9. Preissl, R., Wichmann, N., Long, B., Shalf, J., Ethier, S., Koniges, A.: Multithreaded global address space communication techniques for gyrokinetic fusion applications on ultra-scale platforms. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11 (2011)
10. Sourcery Institute: Hello-world (2022). https://github.com/sourceryinstitute/hello-world
11. Sourcery Institute (2022). https://github.com/sourceryinstitute/dag
12. Bauer, L., Grudnitsky, A., Shafique, M., Henkel, J.: PATS: a performance aware task scheduler for runtime reconfigurable processors. In: 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, pp. 208–215. IEEE (2012)
13. Basilio, B., Fraguela, B.B., Andrade, D.: The new UPC++ DepSpawn high performance library for data-flow computing with hybrid parallelism. In: International Conference on Computational Science (2022)
14. Song, F., YarKhan, A., Dongarra, J.: Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–11. IEEE (2009)
15. Cann, D.: Retire Fortran? A debate rekindled. Commun. ACM **35**(8), 81–89 (1992)
16. Dijkstra, E.W.: How do we tell truths that might hurt? ACM Sigplan Notices **17**(5), 13–15 (1982)

17. Kedward, L.J., et al.: The state of fortran. Comput. Sci. Eng. **24**(2), 63–72 (2022)
18. Ehlert, S., et al.: Fortran package manager. In: International Fortran Conference 2021, Zurich, Switzerland, hal-03355768, v1 (2021). https://hal.archives-ouvertes.fr/hal-03355768
19. Austin, B., et al.: NERSC-10 Workload Analysis (2020). https://doi.org/10.25344/S4N30W
20. Fanfarillo, A., Burnus, T., Cardellini, V., Filippone, S., Nagle, D., Rouson, D.: OpenCoarrays: open-source transport layers supporting coarray Fortran compilers, In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, pp. 1–11 (2014)
21. Singleterry, R. C., et al.: OLTARIS: on-line tool for the assessment of radiation in space. In: NASA/TP-2010-216722 (2010). http://oltaris.nasa.gov