SN Computer Science (2024) 5:354
https://doi.org/10.1007/s42979-024-02682-y

ORIGINAL RESEARCH

Scheduling and Performance of Asynchronous Tasks in Fortran 2018 with FEATS

Brad Richardson[1,2] · Damian Rouson[1,2] · Harris Snyder[2] · Robert Singleterry[3]

Received: 30 October 2023 / Accepted: 4 February 2024

Abstract

Most parallel scientific programs contain compiler directives (pragmas) such as those from OpenMP (Hermanns in Parallel programming in Fortran 95 using openMP, 2002. School of Aeronautical Engineering, Universidad Politécnica de Madrid, España, 2011), explicit calls to runtime library procedures such as those implementing the Message Passing Interface (MPI) (in A message-passing interface standard version 4.0, 2021. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf), or compiler-specific language extensions such as those provided by CUDA (Ruetsch and Fatica in CUDA Fortran for scientists and engineers: best practices for efficient CUDA Fortran programming, Elsevier, 2013). By contrast, the recent Fortran standards empower developers to express parallel algorithms without directly referencing lower-level parallel programming models (Numrich in Parallel programming with co-arrays, CRC Press, 2018, and Curcic in Modern Fortran: building efficient parallel applications, Manning Publications, 2020). 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

This article is part of the topical collection "Applications and Frameworks using the Asynchronous Many Task Paradigm" guest edited by Patrick Diehl, Hartmut Kaiser, Peter Thoman, Steven R. Brandt, and "Ram" Ramanujam.

✉ Brad Richardson
brad.richardson@lbl.gov

Damian Rouson
rouson@lbl.gov

Harris Snyder
harris@archaeologic.codes

Robert Singleterry
robert.c.singleterry@nasa.gov

1 Lawrence Berkeley National Laboratory, Berkeley, CA, USA

2 Archaeologic, Inc., Oakland, CA, USA

3 NASA Langley Research Center, Hampton, VA, USA

Published online: 28 March 2024

Introduction

Modern computing hardware has evolved to offer a variety of opportunities to exploit parallelism for high performance—including multicore processors with 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, sets of images 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 to be a larger challenge 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.

Ultimately, the goal is reduce the time to solution, increase the reliability of the solution, utilize state-of-the-art hardware, and increase the solution's maintainability over the lifetime of the solution. Fortran and FEATS allow for a task parallel solution that hits these marks.

Implementation

FEATS is designed around the use of Fortran coarrays to provide distributed multiprocessing and data exchange between application images. 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.

Tasks have inputs and outputs, so there must be a 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 needs not directly deal with any details related to parallel programming or even understand coarrays. The "execute" function of each task accepts an array of `payload_t` objects, the results of each task on which it depends, and returns a single `payload_t` object result. 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. Additionally, coarray elements cannot contain polymorphic components. 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 statically sized elements will work correctly, whereas one with pointers and allocatable components likely will not). Alternately, a string representation can be used/is supported for the serialization and deserialization. Although arguably an esthetically "inelegant" approach, the authors see it as an acceptable engineering tradeoff in the interest of generality.

The tasks are organized as a DAG. This is stored as an array of vertices, where each vertex contains a task, and an array of integers identifying the tasks on which it depends. We note that in theory, it would be possible for a single image to construct the DAG, and for FEATS to use `co_broadcast` to send it to the remaining images, but not all compilers have implemented the functionality to allow `co_broadcast` of objects with polymorphic components, so in practice, every image must provide exactly the same DAG to FEATS for execution. It is the responsibility of the framework user to define the derived types representing each type of task for their application and to implement the logic for defining the task DAG. Once the DAG is defined, the FEATS framework executes it. The scheduler implementation is provided by the framework. It should be noted that this does require that the entire DAG be defined prior to the execution of any tasks.

Scheduling Task Execution

Two different algorithms for scheduling task execution have been implemented and measured for performance. The algorithms are described in the following subsections.

An Explicit Scheduler Image

This algorithm designates one image, the scheduler, responsible for assigning tasks for the remaining images, executors, to execute. The general algorithm performed by the scheduler image is as follows.

- Find an executor that has posted it is ready
  - While it does this, it keeps track of what tasks have been completed
- Find an uncompleted task with all dependencies completed
- "Wait" for the ready executor (balances posts/waits)
- Assign the task to the executor
- Post that the executor has been assigned a task
- Repeat

The general algorithm performed by an executor is as follows.

- Post ready for a task
- Wait until it has been assigned a task
- Collect payload outputs from executors that ran dependent tasks
  - It accesses the history kept by the scheduler to determine where the outputs reside
- Execute task and store result in its payload mailbox
- Repeat

First to Claim

This algorithm requires no scheduler image, but rather puts all executors on equal footing with respect to claiming and executing tasks. The general algorithm performed by each executor is as follows.

- Find task that has not been claimed, and all of its dependencies have been completed
- Attempt to claim that task
  - Another executor may have claimed it by now
- Collect payload outputs from executors that ran dependent tasks
  - "Wait" on event that it was completed
- Execute task and store result in its payload mailbox
- Post to all executors that the task has been completed and increment task completed counter
- Repeat

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 serves as impediments to the desired features of the framework.

Advantages

There are several features of the modern Fortran language that makes 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 is completed before subsequent tasks begin. The coarray feature of Fortran provides a simple and effective method of performing one-sided communication between 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 are 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 case of large tasks DAGs 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 executors operating on partitions of the task DAG.

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 DAG 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 DAG.

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.

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 that 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 executor must have a complete copy of the DAG and its tasks, because the tasks themselves cannot be communicated to the executors later. This represents a moderate inefficiency in data storage and in initial execution for each executor to compute/construct the DAG. 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. The Fortran standard committee has accepted this as an item to address for the upcoming F202Y revision.

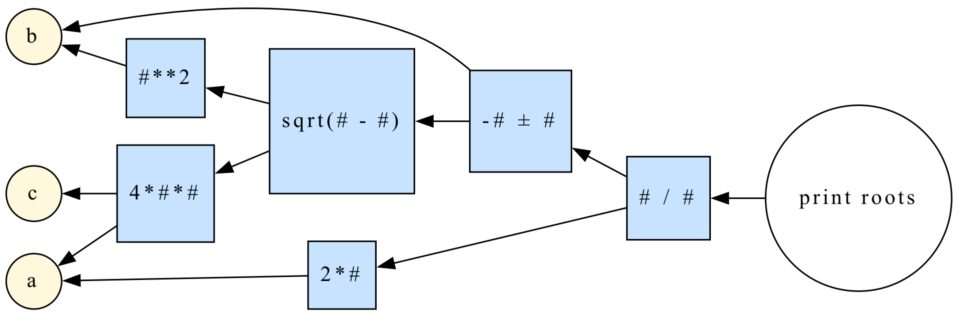


Fig. 1 Graphical representation of the computational tasks involved in calculating the roots of a quadratic equation

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 by one executor, 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 executors.

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 DAG 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 purpose of demonstrating the examples shown below. Additionally, we had to simplify the code, remove the use of external dependencies, and work around various internal compiler errors in order to get the examples shown to compile and execute with the remaining compilers.

Examples

The examples described in this section can be found in the FEATS repository at https://github.com/sourceryinstitute/feats.

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 that the tasks have been appropriately defined.

```
solver = &
    dag_t([ &
          vertex_t([integer::], a_t(a)) &
        , vertex_t([integer::], b_t(b)) &
        , vertex_t([integer::], c_t(c)) &
        , vertex_t([2], b_squared_t()) &
        , vertex_t([1, 3], four_ac_t()) &
        , vertex_t([4, 5], square_root_t()) &
        , vertex_t([2, 6], minus_b_pm_square_root_t()) &
        , vertex_t([1], two_a_t()) &
        , vertex_t([8, 7], division_t()) &
        , vertex_t([9], printer_t()) &
    ])
```

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
```

LU Decomposition

LU decomposition is a common, computationally intensive operation. It involves finding the lower (L) and upper (U) triangular matrices that when multiplied together result in the original matrix. By breaking the task down into appropriate steps, we can define a DAG to perform the operation. The general algorithm is as follows in pseudocode.

```
matrix(1, :, :) = initial_matrix
L = 0
for i = 1, num_rows; L(i, i) = 1
step = 1, num_rows-1
  row = step+1, num_rows
     L(row, step) =
            matrix(step, row, step)
          / matrix(step, step, step)
     matrix(step+1, row, :) =
            matrix(step, row, :)
          - matrix(step, row, :) * L(row, step)
U = matrix(num_rows, :, :)
```

The code to transform the above algorithm into a task DAG executable by FEATS is as follows. It prints the initial matrix, as well as the intermediate matrix at the completion of each step. The final matrix printed is the L matrix, and the penultimate matrix is the U matrix.

```
num_tasks = &
    4 &
    + (matrix_size-1)*2 &
    + sum([((matrix_size-step)*3, step=1,matrix_size-1)])
allocate(vertices(num_tasks))

vertices(1) = vertex_t([integer::], initial_t(matrix))
vertices(2) = vertex_t([1], print_matrix_t(0))
do step = 1, matrix_size-1
  do row = step+1, matrix_size
    latest_matrix = &
        1 &
        + sum([(3*(matrix_size-i), i = 1, step-1)]) &
        + 2*(step-1)
    task_base = &
        sum([(3*(matrix_size-i), i = 1, step-1)]) &
        + 2*(step-1) + 3*(row-(step+1))
    vertices(3+task_base) = vertex_t( &
        [latest_matrix], &
        calc_factor_t(row=row, step=step))
    vertices(4+task_base) = vertex_t( &
        [latest_matrix, 3+task_base], &
        row_multiply_t(step=step))
    vertices(5+task_base) = vertex_t(&
        [latest_matrix, 4+task_base], &
        row_subtract_t(row=row))
  end do
  reconstruction_step =
      3 &
      + sum([(3*(matrix_size-i), i = 1, step)]) &
      + 2*(step-1)
  vertices(reconstruction_step) = vertex_t( &
      [ 1 &
        + sum([(3*(matrix_size-i), i = 1, step-1)]) &
        + 2*(step-1) &
      , [(5 &
        + sum([(3*(matrix_size-i), i = 1, step-1)]) &
        + 2*(step-1) &
        + 3*(row-(step+1)) &
          , row=step+1, matrix_size)] &
      ], &
```

```
      reconstruct_t(step=step))
  ! print the just reconstructed matrix
  vertices(reconstruction_step+1) = vertex_t( &
      [reconstruction_step], print_matrix_t(step))
end do
vertices(num_tasks-1) = vertex_t( &
    [([( 3 &
      + sum([(3*(matrix_size-i), i = 1, step-1)]) &
      + 2*(step-1) + 3*(row-(step+1)) &
        , row=step+1, matrix_size)] &
      , step=1, matrix_size-1)] &
    , back_substitute_t(n_rows=matrix_size))
vertices(num_tasks) = vertex_t( &
    [num_tasks-1], print_matrix_t(matrix_size))

dag = dag_t(vertices)
```

For a 3×3 matrix, the above code creates a task DAG like the one shown in Fig. 2. Execution of the task DAG produces output like shown below.

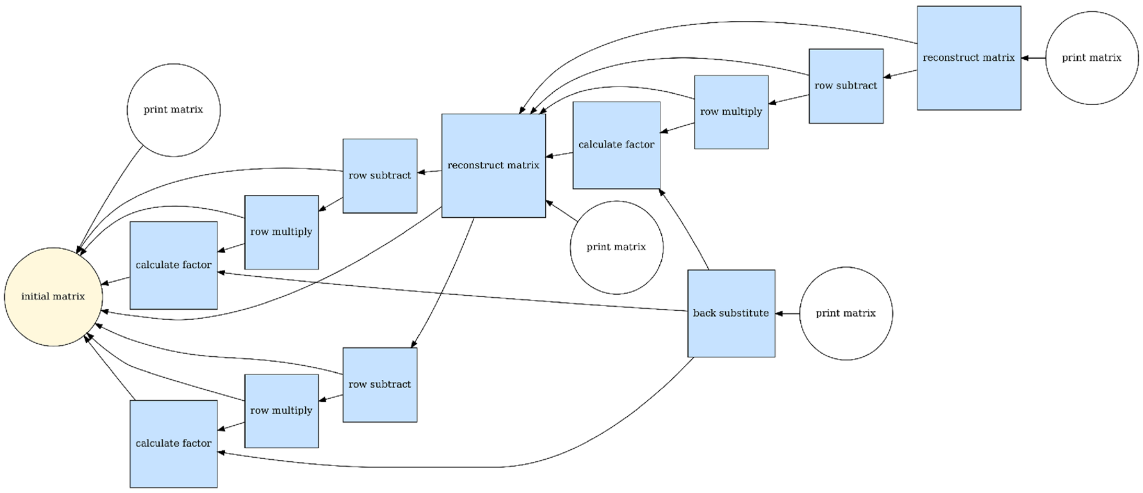


Fig. 2 Task graph of LU decomposition for a 3x3 matrix

```
Step:  0
   11.052          411.77          573.46
   9.7038          978.47          751.12
   564.47          892.35          806.38

Step:  1
   11.052          411.77          573.46
   0.0000          616.92          247.60
   0.0000         −20139.         −28483.

Step:  2
   11.052          411.77          573.46
   0.0000          616.92          247.60
   0.0000          0.0000         −20401.

Step:  3
   1.0000          0.0000          0.0000
  0.87804          1.0000          0.0000
   51.075         −32.644          1.0000
```

Performance

Using the problem of finding the LU decomposition of a 100×100 matrix, the performance of both task scheduling methods was evaluated. See sections 2.1.1 (explicit scheduler image, ESI) and 2.1.2 (first to claim, FTC) for an overview of the two strategies. Performance experiments were conducted using two computer systems (a desktop computer equipped with an Intel Core i5-6500 and the Perlmutter supercomputer) and three different Fortran compilers (the NAG compiler version 7.1 Build 7138, gfortran version 13.2.0 on the desktop computer and version 11.2.0 on Perlmutter with OpenCoarrays version 2.10.1, and the Cray compiler version 15.0.1). Experiments using the NAG compiler were done using the compile time flags `-O4 -Onoteams -Orounding -coarray`. Experiments with the Cray compiler and gfortran + opencoarrays used `-O3`. Note that the ESI strategy was implemented such that if there is only one running image, the scheduler image does the work required for each task itself, rather than assigning it to another image (though the scheduling logic still runs in order to choose the next task).

With the NAG compiler, which was only available on the Intel i5 desktop system, moving from the ESI implementation to the FTC implementation reduced the single-image execution time from almost 12 s to under 5 s. Both implementations saw the runtime cut approximately in half by moving to two images, but adding a third or fourth resulted in little change to the execution time (Figs. 3 and 4). On that system, which has four cores, more than 4 images was not expected to improve performance and indeed it did not, with performance getting worse after four images.

Gfortran with OpenCoarrays fared quite differently from the NAG compiler. On the i5 system, the ESI strategy finished with less than half of the average runtime of the FTC implementation, in a reversal of the performance of the two algorithms under the NAG compiler. Scaling to multiple images resulted in *longer* execution times than with a single image (Figs. 5 and 6). This was true for both algorithms, though the ESI approach was dramatically worse. Using multiple images also failed to produce a speedup on

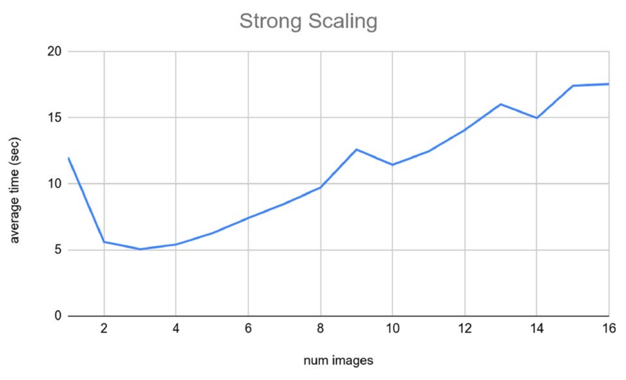


Fig. 3 Intel i5, NAG Compiler, Explicit Scheduler Image

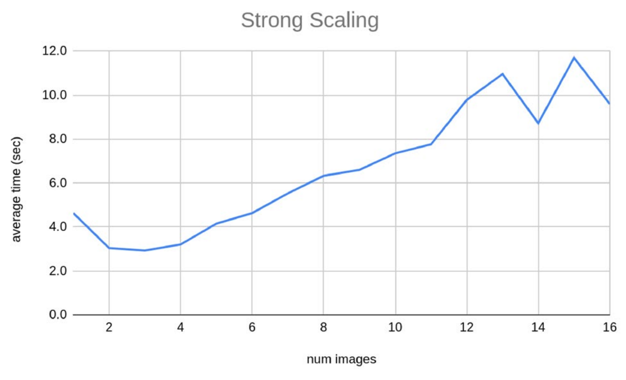


Fig. 4 Intel i5, NAG Compiler, First to Claim

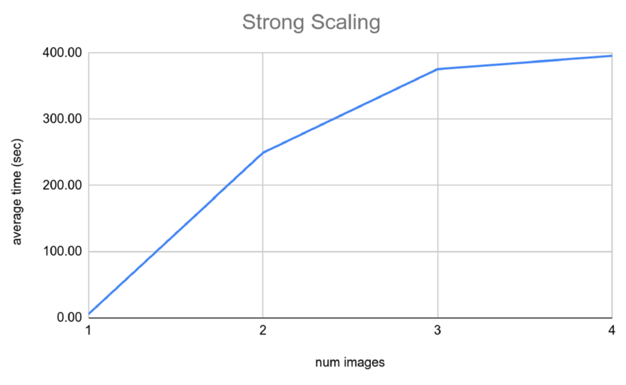


Fig. 5 Intel i5, Gfortran with OpenCoarrays, Explicit Scheduler Image

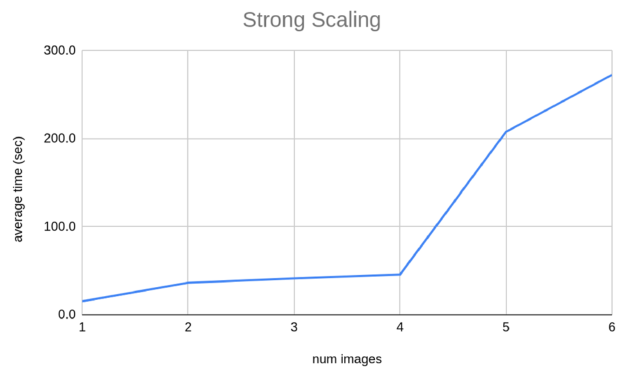


Fig. 6 Intel i5, Gfortran with OpenCoarrays, First to Claim

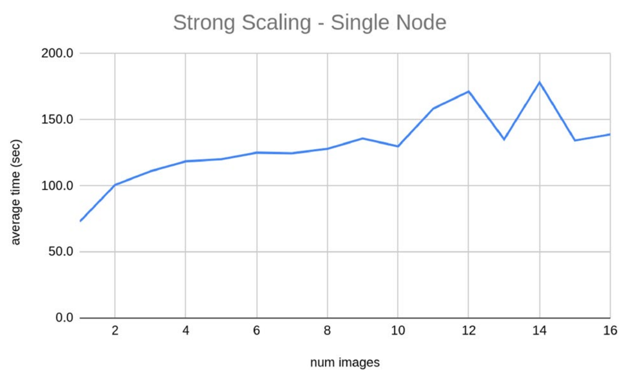


Fig. 7 Perlmutter (1 node), Gfortran with OpenCoarrays, First to Claim

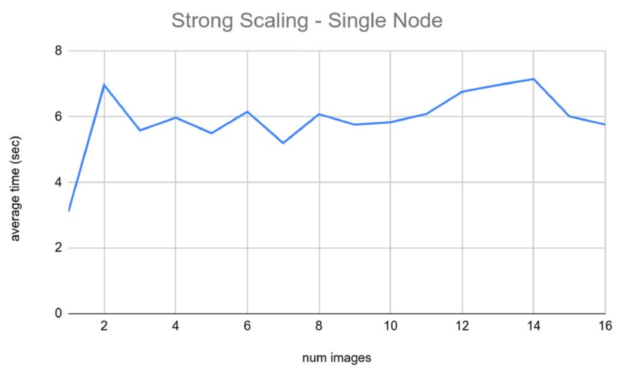


Fig. 8 Perlmutter (1 node), Cray compiler, Explicit Scheduler Image

Perlmutter, at least with the FTC strategy (Fig. 7); the ESI strategy was not tried.

The Cray compiler was only available on Perlmutter. The ESI strategy ran with about a quarter the runtime of the FTC in the single-image case. However, the ESI strategy failed to scale; using two or more images (on the same node) doubled the execution time, which remained roughly constant with number of images thereafter (Figs. 8 and 9). The FTC strategy, while slower in the single image case, did produce reductions in execution time for adding additional images until 10 images were reached.

Scaling was also assessed on Perlmutter (using the Cray compiler) when the images are on different physical nodes.

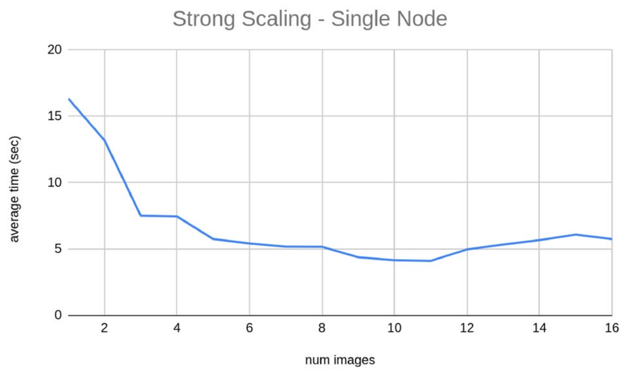


Fig. 9 Perlmutter (1 node), Cray compiler, First to Claim

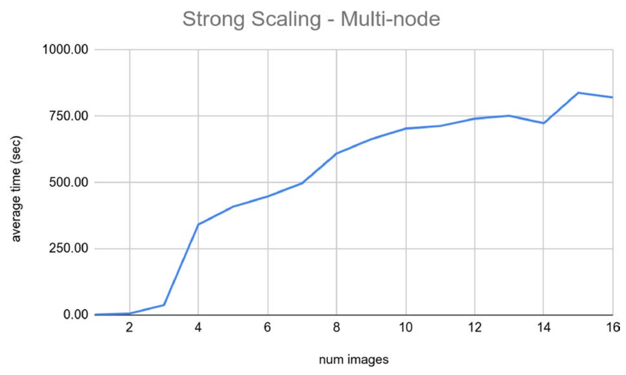
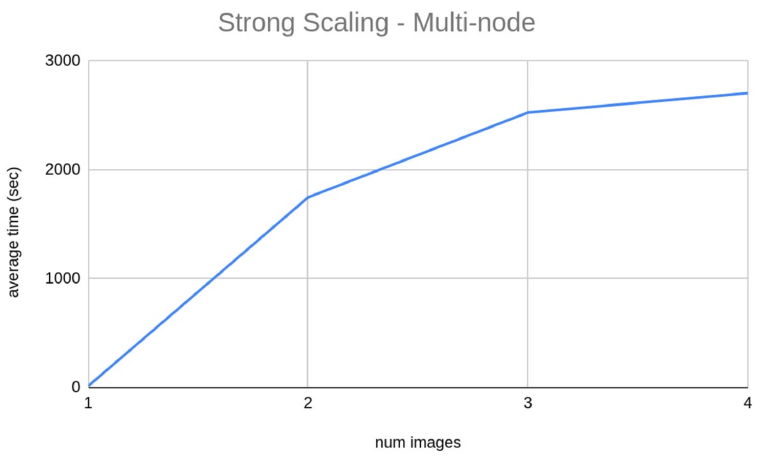


Fig. 10 Perlmutter (multinode), Cray compiler, Explicit Scheduler Image

Fig. 11 Perlmutter (multinode), Cray compiler, First to Claim



The result for both strategies was that using multiple nodes dramatically *increases* total runtime (Figs. 10 and 11).

The results from these experiments are not encouraging. Admittedly, breaking down the LU decomposition of a 100×100 matrix into 15,000 distinct tasks does make for something of a toy problem, but regardless of whether this example problem is well suited to scaling to many images in the first place, the experiments conducted suggest that some of the parallel features in Fortran 2018 do not enjoy performance portability. The specific details of how coarray communication and events are implemented by a given compiler runtime clearly matter a great deal, which makes it difficult for programmers to utilize them effectively. Further investigation is needed to determine the exact cause of the performance pathologies exhibited by certain compiler/algorithm combinations.

Conclusion

We believe that 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 toward providing such a capability to Fortran users.

We look forward to working on several unresolved issues in FEATS. 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 the applicability of different scheduling algorithms to various types of applications and compare the performance characteristics of FEATS with other task scheduling implementations. 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 work-stealing with the Intermediate Complexity Atmospheric Research model [8].

Supplementary Materials

The code for all examples is available on GitHub®.[1]

Funding The authors would like to thank NASA Langley Research Center for providing the initial funding for this work.

Declarations

Conflict of interests The authors declare that they have no competing interests.

Informed consent Not applicable, since human research subjects were not used.

Research involving human participants and/or animals This article does not contain any studies with human participants performed by any of the authors.

References

1. Hermanns M. Parallel programming in Fortran 95 using openMP, 2002. School of Aeronautical Engineering: Universidad Politécnica de Madrid, España; 2011.
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. Ruetsch G, Fatica M. CUDA Fortran for scientists and engineers: best practices for efficient CUDA Fortran programming. Elsevier; 2013.
4. Numrich RW. Parallel programming with co-arrays. CRC Press; 2018.
5. Curcic M. Modern Fortran: building efficient parallel applications. Manning Publications; 2020.
6. Mattson TG, Sanders B, Massingill B. Patterns for parallel programming. Pearson Education; 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. 2015;29(3):261–73.
8. Gutmann E, Barstad I, Clark M, Arnold J, Rasmussen R. The intermediate complexity atmospheric research model (ICAR). J Hydrometeorol. 2016;17(3):957–73.
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; 2011. pp. 1–11.
10. Hello-world. Sourcery Institute. https://github.com/sourceryinstitute/hello-world
11. DAG. Sourcery Institute. 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. IEEE; 2012. pp. 208–215.
13. Fraguela BB, Andrade D. The new UPC++ DepSpawn high performance library for data-flow computing with hybrid parallelism. In: International conference on computational science. Springer; 2022. pp. 761–774.
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; 2009. pp. 1–11.
15. Cann D. Retire Fortran? A debate rekindled. Commun ACM. 1992;35(8):81–9.
16. Dijkstra EW. How do we tell truths that might hurt? ACM Sigplan Not. 1982;17(5):13–5.
17. Kedward LJ, Aradi B, Čertík O, Curcic M, Ehlert S, Engel P, Goswami R, Hirsch M, Lozada-Blanco A, Magnin V, et al. The state of Fortran. Comput Sci Eng. 2022;24(2):63–72.
18. Ehlert S, Čertík O, Curcic M, Jelínek J, Kedward L, Magnin V, Pagone E, Richardson B, Urban J. Fortran package manager. In: International Fortran conference 2021; 2021.
19. NERSC-10 Workload Analysis (Data from 2018). NERSC. 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; 2014. pp. 1–11.
21. Singleterry RC Jr, Blattnig SR, Clowdsley MS, Qualls GD, Sandridge CA, Simonsen LC, Slaba TC, Walker SA, Badavi FF, Spangler JL, et al. Oltaris: on-line tool for the assessment of radiation in space. Acta Astronaut. 2011;68(7–8):1086–97.

[1] https://github.com/sourceryinstitute/FEATS.