Development and performance comparison of MPI and Fortran Coarrays within an atmospheric research model

Extended Abstract

Soren Rasmussen[1], Ethan D Gutmann[2], Brian Friesen[3], Damian Rouson[4], Salvatore Filippone [1], Irene Moulitsas [1]

[1]Cranfield University, UK
[2]National Center for Atmospheric Research, USA
[3]Lawrence Berkeley National Laboratory, USA
[4]Sourcery Institute, USA

ABSTRACT

A mini-application of The Intermediate Complexity Research (ICAR) Model offers an opportunity to compare the costs and performance of the Message Passing Interface (MPI) versus coarray Fortran, two methods of communication across processes. The application requires repeated communication of halo regions, which is performed with either MPI or coarrays. The MPI communication is done using non-blocking two-sided communication, while the coarray library is implemented using a one-sided MPI or OpenSHMEM communication backend. We examine the development cost in addition to strong and weak scalability analysis to understand the performance costs.

CCS CONCEPTS

• **Software and its engineering** → **Parallel programming languages**; • **Applied computing** → *Environmental sciences*;

KEYWORDS

coarray Fortran, message passing interface, computational hydrometeorology

ACM Reference Format:
Soren Rasmussen[1], Ethan D Gutmann[2], Brian Friesen[3], Damian Rouson[4], Salvatore Filippone [1], Irene Moulitsas [1] [1]Cranfield University, UK [2]National Center for Atmospheric Research, USA [3]Lawrence Berkeley National Laboratory, USA [4]Sourcery Institute, USA . 2018. Development and performance comparison of MPI and Fortran Coarrays within an atmospheric research model. In *Proceedings of PAW-ATM 18: Parallel Applications Workshop, Alternatives to MPI, Dallas, TX, USA, November 11–16, 2018 (PAW18),* 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
PAW18, November 11–16, 2018, Dallas, TX, USA
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Motivation and Background

In high performance computing MPI has been the de facto method for memory communication across a system's nodes for many years. MPI 1.0 was released in 1994 and research and development has continued across academia and industry. A method in Fortran 2008, known as coarray Fortran, was introduced to express the communication within the language [5]. This work was based on an extension to Fortran that was introduced by Robert W. Numrich and John Reid in 1998 [7]. Coarray Fortran, like MPI, is a single-program, multiple-data (SPMD) programming technique. Coarray Fortran's single program is replicated across multiple processes, which are called "images". Unlike MPI, it is based on the Partitioned Global Address Space (PGAS) parallel programming model. This allows the Fortran syntax to easily express communication while maintaining the transparency of the underlying algorithm concept. This will be further discussed in the programmability section.

The application used to examine the different programming modules is a mini-application of The Intermediate Complexity Atmospheric Research (ICAR) model. This simplified atmospheric model was developed at the National Center for Atmospheric Research (NCAR) to predict aspects of weather such as precipitation, temperature, and humidity [3]. The main impetus of the investigation is to understand the scalability and performance of the different coarray and MPI programming models. The ICAR mini-app was originally developed using coarrays to communicate halo regions. For this paper we modified the existing code to use MPI, instead of coarrays, for communication between processes.

We used Open Coarrays, a library implementation of coarray Fortran, for our runtime comparisons. The Open Coarrays communication backend can be implemented with either an OpenSHMEM layer or MPI. Open Coarrays' MPI implementation uses one-sided communication with passive synchronization [2]. This has allowed us to do performance comparisons between three versions of the ICAR mini-app: the OpenSHMEM backend, the coarray one-sided MPI, and the two-sided MPI implementation.

Past work has been done on the scalability and performance differences between coarrays and MPI in the past [1, 4] . Past experiments using this specific mini-app have looked at the comparisons between the OpenSHMEM communication and the MPI communication backend [8]. To our knowledge the work done here is

unique because we are using a higher number of processes then past coarray vs. MPI comparisons. Additionally we are comparing a MPI version and two different coarray communication backends, one which itself is an MPI implementation.

1.2 Programmability

In the ICAR mini-app the domain is split from a rectangular cuboid across the x and y-axis. Therefore the process/image can have up to four neighbors it needs to send and receive boundary regions from. This halo region commutation is done with coarrays or MPI and offers a useful comparison in the ease of programmability. In the coarray fortran model it is easy to express movement between different images. The following code shows how the `east_neighbor` would transfer the halo region of array `A` to the `west_neighbor`.

```
east_halo(1:halo_size,:,1:ny)[west_neighbor]
    = A(start:start+halo_size, :,:)
```

The single line that `east_neighbor` would run in the coarray model, requires an implementation of `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait` in two MPI ranks. Data structures need to be created or modified to pass around the `MPI_Request` handles so `MPI_Wait` knows when the communication is complete.

In our application the complexity of MPI is further increased when sending a large number of subarrays. A very natural translation of the above communication pattern of subarrays would be as follows:

```
call MPI_Isend(A(start:start+halo_size,:,:), &
    length, MPI_Real, west_neighbor, tag, &
    MPI_COMM_WORLD, send_request, ierr)
```

In the context of our mini-application, this communication has to occur on up to four sides, with nine different arrays for variables such as rain mass, water vapor, etc. being passed. The above code is a natural MPI transposition, but experienced MPI programmers will have noticed that dealing with (possibly non-contiguous) portions of a multidimensional array implies that the Fortran compiler would sometimes create a temporary copy of the subarray to be passed to the `MPI_Isend` and `MPI_Irecv` calls. However the MPI programming model requires the communication buffers to be persistent until the completion by `MPI_Wait`; by that time, the temporary copy of the subarray would often have disappeared. This application leads to segmentation faults whose proper diagnosis can be quite tricky, even with all debugging flags turned on. It is an error that domain scientists without in-depth knowledge of MPI would have trouble solving. `MPI_Type_create_subarray` and `MPI_Type_commit` are used to ensure the data transfers occur correctly; for efficiency reasons, since the types do not change during the simulation, they can be defined once and then cached. Thus one line of communication using coarrays now requires five MPI calls and a modified data structure for a caching scheme of the MPI type encodings, including the send and receive `MPI_Requests` for `MPI_Wait`.

In Figure 2 we attempt to quantify the amount of additional lines of code that need to be written. An "additional line" is defined as an additional line of code that is not found in the other implementation. If a line is changed to a different function call, such as 'MPI_Block' instead of 'sync all', it is not counted as an additional line. If a single function call runs multiple lines due to the number of arguments, it is treated as one line. As to be expected our physics code, 'mp_thompson', and the domain files did not need significant changes, since the communication is handled in the exchange files. In those exchange files 48.65% and 91.13% more additional code needed to be written in the interface and implementation file. It is quite clear that the ease of programming enabled by Coarray Fortran enables the application developer to concentrate on less mundane topics, leaving the handling of such low-level details to the compiler and the runtime library.

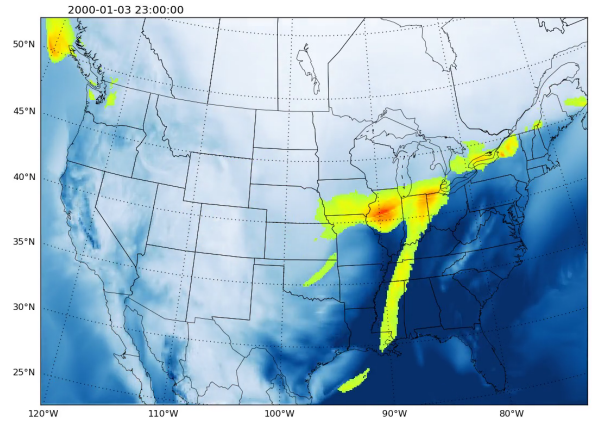


Figure 1: A visualization of the atmospheric distribution of water vapor (blues) and the resulting precipitation (green to red) simulated by The Intermediate Complexity Atmospheric Research (ICAR).

File	Additional lines	% of additional code
mp_thompson	15	0.28
domain_implementation	6	1.09
domain_interface	0	0
exchange_interface	54	48.65
exchange_implementation	226	91.13

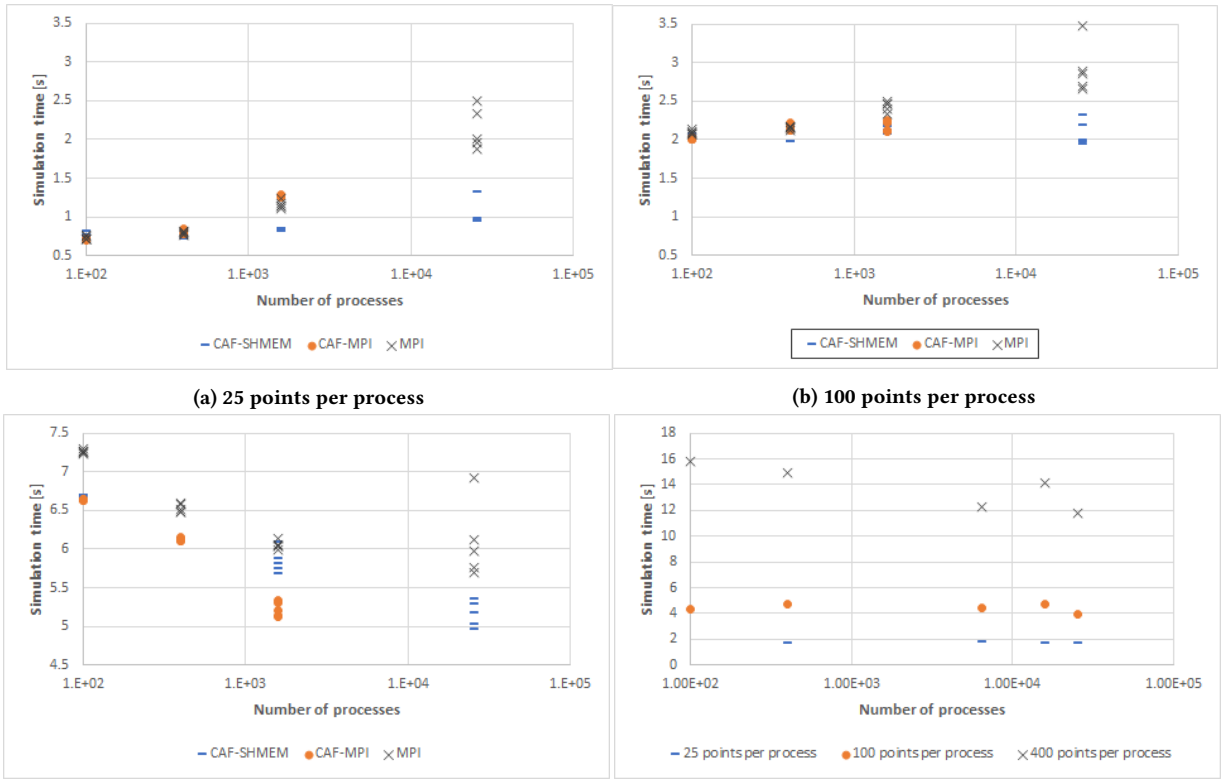
Figure 2: Number of additional lines that need to be written. A changed line does not count towards the total.

3 METHODOLOGY

3.1 Compilers, runtimes, and hardware

The experiments were done on two different systems, the first being NCAR's system Cheyenne, which is a SGI ICE XA Cluster [6]. It has 4032 computation nodes and each node is dual socket with 2.3-Ghz Intel Xeon E5-2697V4 processors. The interconnect is a Mellanox EDR Infiniband with a partial 9D Enhanced Hypercube single-plane topology. The compiler we used for comparison is GNU's gfortran 6.3 with an application binary interface of coarrays from OpenCoarrays 1.9.4 [2]. The MPI implementation used for both the MPI and OpenSHMEM runs was version 2.15f of SGI's Message Passing Toolkit (MPT), the preferred version on Cheyenne.

2 DISCUSSION OF RESULTS



(a) 25 points per process

(b) 100 points per process

(c) 400 points per process

(d) Cray weak scaling

Figure 3: (a-c) Weak scaling results for 25, 100, and 400 points per process (d) weak scaling for Cray.

The second system used was Lawrence Berkeley National Laboratory's (LBNL) Cori, a Cray XC40 with 12,076 total compute nodes [8]. Of those nodes, 9688 of them are single-socket, 68-core Intel Xeon Phi Processor 7250 ("Knight's Landing") at 1.4 GHz. We used Knight's Landing with the Cray Compiling Environment (CCE) 8.7.1. The Cray compiler uses Cray's proprietary PGAS runtime for implementing coarrays within Fortran. It was run with 2 MiB hugepages enabled, rather than the default 4 KiB pages, because huge pages often gives better performance for PGAS codes. For the runs done on both Cheyenne and Cori a single core was used per MPI rank or coarray image.

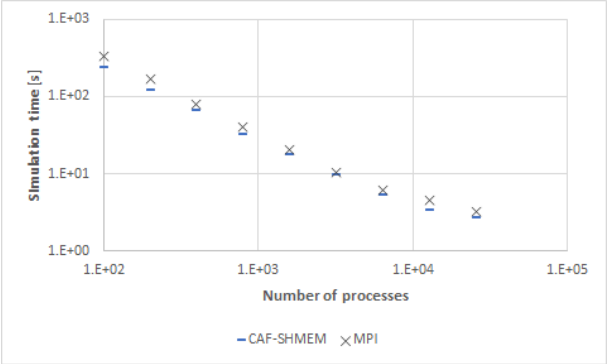
4 RESULTS

In figures 3a, 3b, and 3c respectively, the results are presented for weak scaling at 25, 100, and 400 points per process. These runs were all done on the SGI cluster Cheyenne. At each problem size we gathered multiple timing samples of the coarray version with the OpenSHMEM communication backend, the coarray version with the MPI backend, and the plain MPI implementation. At lower numbers of points per process the OpenSHMEM communication backend performs better. As the number of points per process increases, OpenSHMEM continues to perform better but the pure MPI version keeps pace. For the weak scaling runs done using Cray's proprietary PGAS runtime on Cori 3d, the results are good. There is no noticeable deterioration in efficiency, meaning the parallel overhead is not slowing down the runs.

It is interesting to note that the MPI implementation had the largest amount of variance for any one run. For the OpenSHMEM runs the variance was always under 0.4 seconds while for the pure MPI runs the smallest was 0.63 seconds and the largest 1.2. For 25, 100, and 400 points per process the variance was 39%, 19%, and 8% for OpenSHMEM and 34%, 31%, 22% for the pure MPI. A data trend that was unclear is the decrease in simulation time for the the first few weak scaling runs for 400 points per processor. This occurred in all three of the different implementations.

For strong scaling 4 we used up to 25,600 processes and found that at every data point OpenSchmem was outperforming MPI. At high number of processes we were unable to get the the coarray Fortran MPI communication backend to work.

The coarray Fortran with MPI backend stopped being usable as we went over 2,000 processes; strictly speaking it did not stop working, but the initialization time started to increase exponentially. At 2,000 processes it would take about an hour to start the process and at 3,000 processes it exceeded the 12 hour wall clock limit. Further investigation will be needed to understand why this is occurring and fix it.



(a) Strong scaling results

Figure 4: Strong scaling results for 2000 x 2000 x 20 problem

5 CONCLUSIONS AND FUTURE WORK

We have shown that the easy syntax provided by coarray Fortran can be exploited without any serious effects on the runtime and scalability. Coarray syntax allows domain-scientists to take advantage of high performance computing resources and MPI without unduly burdening them with the details. The compiler and coarray library will also handle any changes to those details as computing hardware and the MPI standard advances. One can surmise this would allow them to focus more on the science and applications of their expertise. An advantage of using the OpenCoarray library is that it takes no extra coding effort to switch between the MPI and OpenSHMEM communication backend. There might be circumstances where the MPI backend will be faster and having the ability to easily switch is advantageous.

For future work we would like to do runs with a higher processor count to better understand where the strong scaling and the Crays weak scaling tail off. Running the scaling tests with different MPI implementations would reveal if one would be able to match the OpenSHMEM timing. An interesting question is how manually packing MPI buffers before sending them would effect performance in relation to the one-sided coarray version and the two-sided `MPI_Type_create_subarray` version. Would the MPI implementation's handling of memory management be more efficient than a user's manually packed buffers?

The variance in the weak scaling runs of the pure MPI implementation is also another area of possible interest. Both OpenSHMEM and the MPI implementation's variance decreased as the number of processes used increased, but at 22% MPI remained fairly high. Each process count was only run five times for each version, it would interesting to see if this variance is truly high and how it is effected by more runs and a higher process count.

Within the ICAR mini-app the segment of code where the heavy computation occurs has been written with OpenMP directives. While they were not used for this analysis we plan on investigating how OpenMP would effect the runtime and efficiency. This study would help indicate how to effectively use the cores allotted when running the larger ICAR application.

ACKNOWLEDGMENTS

This paper is based upon work supported by NSF's National Center for Atmospheric Research, a major facility fully funded by the National Science Foundation.

We would like to acknowledge high-performance computing support from Cheyenne (doi:10.5065/D6RX99HX) provided by NCAR's Computational and Information Systems Laboratory, sponsored by the National Science Foundation.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] John V Ashby and John K Reid. 2008. Migrating a scientific application from MPI to coarrays. *CUG 2008 Proceedings. RAL-TR-2008-015: ftp://ftp. numerical. rl. ac. uk/pub/reports/arRAL2008015. pdf* (2008).

[2] Alessandro Fanfarillo, Tobias Burnus, Valeria Cardellini, Salvatore Filippone, Dan Nagle, and Damian Rouson. 2014. OpenCoarrays: open-source transport layers supporting coarray Fortran compilers. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 4.

[3] Ethan Gutmann, Idar Barstad, Martyn Clark, Jeffrey Arnold, and Roy Rasmussen. 2016. The intermediate complexity atmospheric research model (ICAR). *Journal of Hydrometeorology* 17, 3 (2016), 957–973.

[4] Manuel Hasert, Harald Klimach, and Sabine Roller. 2011. Caf versus mpi-applicability of coarray fortran to a flow solver. In *European MPI Users' Group Meeting*. Springer, 228–236.

[5] ISO/IEC 1539-1:2010 2010. *Information technology – Programming languages – Fortran – Part 1: Base language*. Standard. International Organization for Standardization, Geneva, CH.

[6] MultiMedia LLC. 2018. Cheyenne. (2018). https://www2.cisl.ucar.edu/resources/computational-systems/cheyenne

[7] Robert W. Numrich and John Reid. 1998. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum* 17, 2 (Aug. 1998), 1–31. https://doi.org/10.1145/289918.289920

[8] Damian Rouson, Ethan D. Gutmann, Alessandro Fanfarillo, and Brian Friesen. 2017. Performance Portability of an Intermediate-complexity Atmospheric Research Model in Coarray Fortran. In *Proceedings of the Second Annual PGAS Applications Workshop (PAW17)*. ACM, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/3144779.3169104