# Solving Sparse Linear Systems of Equations using Fortran Coarrays

Ambra ABDULLAHI HASSAN [a], Valeria CARDELLINI [a] and Salvatore FILIPPONE [b]

[a] *University of Rome Tor Vergata*
[b] *Cranfield University*

**Abstract.** Coarrays have been part of the Fortran standard since Fortran 2008 and provide a syntactic extension of Fortran to support parallel programming, often called Coarray Fortran (CAF). Although MPI is the de facto standard for parallel programs running on distributed memory systems and few scientific softwares are written in CAF, many scientific applications could benefit from the use of CAF. We present the migration from MPI to CAF of the libraries PSBLAS and MLD2P4 for the solution of large systems of equations using iterative methods and preconditioners. In this paper we describe some investigations for implementing the necessary communication steps in PSBLAS and MLD2P4 and provide performance results obtained on linear systems arising from discretization of 2D and 3D PDEs.

**Keywords.**

## 1. Introduction

Fortran has been the dominant language in High Performance Computing and still is a very important player in the field. In past couple of decades, the language has changed significantly, and now supports class abstractions, object-oriented programming, pure functions, and coarrays [**?**]. Coarrays have been part of the Fortran standard since Fortran 2008 and provide a syntactic extension of Fortran to support parallel programming; in the sequel we will use Fortran instead of CAF to highlight the fact that today the parallelization is an integral part of the language. Following the Partitioned Global Address Space programming model, in Fortran the program is replicated among a certain number of images, executing asynchronously. The PGAS model combines elgance and simplicity, by allowing one-sided communications, and potentially reducing synchronization and code complexity. Coarrays are a language-based approach, meaning that a quality compiler implementation can consider both serial and communication aspects in optimizing the generated code. If the available compiler also provides support for accelerator programming, e.g. by implementing OpenAcc, the programmer can readily get a dual benefit form the combined use of PGAS and accelerator programming models.

The Message Passing Interface (MPI) defines a suite of functions for communication exchange between processes. MPI is library-based: it is in principle independent from any specific compiler, but the user is bound to the vendor's fine-tuning of the MPI library implementation [**?**].

At present MPI is the de facto standard for parallel programs running on distributed memory systems and few scientific softwares are written in CAF; it is our belief that many scientific applications could benefit from the use of the PGAS model implemented in coarrays; to realize the potential, we have to analyze the impact of Coarrays on performance and quality of scientific software.

In this paper we focus on the solution of large and sparse linear systems of equations and we propose the migration of the PSBLAS [**?**] and MLD2P4 [**?**] libraries.

The first one implements various iterative solvers while the second exploits most of the communications subroutines in PSBLAS to implement multigrid preconditioners. Using the combinations of the two libraries, we can compare performances of real-world problems in MPI and in CAF.

Few attempts to implement multigrid solvers in CAF have been made, for example in [**?**] (in which early versions of CAF and MPI have been used). In [**?**] a comparison between MPI3 and CAF is presented for spectral techniques, multigrid techniques and for applications drawn from computational fluid dynamics. It shows that Cray implementations of MPI3 and CAF performances in message passing are comparable (with CAF slightly outperforms MPI3 in some cases), and it points out the advantages of CAF when it comes to ease of implementation. In this paper we compare performances of CAF and MPI on problems arising from discretization of 2D and 3D PDEs, using preconditioned iterative methods (also with multigrid preconditioners). We used the GNU compiler and the OpenCoarray library [**?**], both opensource and easily accessible.

## 2. Fortran Coarrays

Coarrays extend the original Fortran language with minimal syntactic constructs, allowing for Single Program Multi Data (SPMD) parallel programming. It follows the Partitioned Global Address Space (PGAS) parallel programming model, which it is an effective alternative combining the advantages of the SPMD programming style for distributed memory systems with the data referencing semantics of shared memory systems. In the PGAS model, multiple "images" participate into the global address space, each one of them having a local portion and access to the shared portion of the address space. Other languages based on the PGAS model include Unified Parallel C (UPC) [**?**], Chapel [**?**], X10 [**?**] and SHMEM.

With coarrays, a Fortran programmer needs neither subroutines from an external API (as in MPI), nor preprocessing directives (as in OpenMP) to create a parallel programs. Since most of the parallel bookkeeping is handled behind the scene by the compiler, CAF achieves a good trade-off between readability and performance; compared to MPI the parallelization is easier to follow, the code is shorter and less complex. We can think, for example, of the burden of transferring a non-contiguous portion of an array using MPI, a task that is achieved with a single statement, or that in CAF there is no need to write explicit and separate code for sending and receiving a message.

Coarrays are supported by Cray, Intel and GNU compilers; in the GNU compiler the code is translated by emitting calls into an API implemented by the open source opencoarrays project, which in turn provides two different transport layer implementations based on MPI and on GASNet [**?**].

In the Fortran language the program is replicated a certain number of times and each replica, called an image, executes asynchronously with its own set of data objects.

Variables declared as *coarrays* become visible to all images, thereby implementing a locally shared portion of the address space; the syntax uses *coindices* indicated by square brackets, with an appearance very similar to normal array syntax:

```
integer ::co_a[*],co_b(n,m)[*]
```

Any statement with coindices indicates access of the variable on a (possibly) remote image, thus enabling data exchange:

```
co_a[1]=co_a
```

Dropping a coindex implicitly refers to the local version of the variable. Each image is assigned a unique index, through which the programmer can control the flow of the execution. Image indices start from 1 and can be retrieved using the intrinsic function *this_image*(); the total number of images can be retrieved using the function *num_images*(). The exact mechanism to determine how many images will be executed depends on the compiler's implementation: most implementation use a runtime scheme similar to (or directly based on) that of MPI.

Data transfers in Fortran are asynchronous and non-blocking; to prevent deadlock and race conditions, the programmer has to ensure consistency through the use of synchronization statements.

The `SYNC ALL` statement corresponds to a global barrier (all images waiting for each other) while `SYNC IMAGES` allows for synchronization restricted within a subset of images.

In the Fortran 2015 language a number of extensions to the original coarray language have been introduced; among them are the `EVENTS`, which implement an asynchronous notification mechanism. An image can use an `EVENT POST` statement to notify another image without having to go through a full two-way handshaking; in particular, there is no tie to the execution of a matching `EVENT WAIT` by the receiving image . This facility allows for *one-sided* synchronization strategies.

Besides explicit synchronizations, in a Fortran program there can be implicit synchronization points the programmer must be aware of, because they can obviously have a significant impact on performance. An example of an implicit synchronization point is allocation or deallocation of a coarray:

```
integer, allocatable :: co_a(:,:)[:]
allocate(co_a(i,10)[*])
```

Code executed between any two synchronizatoin points can be optimized by the compiler using all common techniques and heuristics as if it was a purely serial code [**?**].

## 3. PSBLAS and MLD2P4

Many scientific and engineering applications require solving a large sparse linear system of equations efficiently on parallel machines; in these cases direct solvers such as Gaussian elimination are often ineffective and iterative solvers are a common alternative.

These techniques produce a sequence of approximate solutions, ideally converging to the real one. Rate of convergence of iterative methods depends upon the condition number of the matrix, hence iterative methods usually involve a transforation of the original matrix into a more well-conditioned one. This transformation is called a preconditioner [**?**]. Krylov methods today the most commonly used iterative techniques; they are based on minimizing the residual $r = b - Ax$ over a certain linear or affine subspace [**?**]. The Conjugate Gradient (CG) [**?**], the Generalized Minimal Residual (GMRES) [**?**] and the Biconjugate Gradient (BiCG) [**?**] are popular Krylov method variants.

Multigrid methods have also shown promising results in solving linear systems arising from partial differential equations and they have the nice property that they can achieve convergence rates which are, in theory, independent of the mesh size [**?**]. Multigrid can be used as stand-alone solvers [**?**] or as preconditioner for a Kylov subspace method [**?**]. PSBLAS is a library of Basic Linear Algebra Subroutineas for parallel sparse applications is designed to handle parallel implementation of iterative solvers for sparse linear systems. It is implemented if object-oriented Fortran 2003; the serial computations are based on an implementation of the serial sparse BLAS originally defined in [**?**], while the inter-process massage exchanges are encapsulated in an application layer based by default on MPI and partially inspired by the Basic Linear Algebra Communication Subroutines (BLACS) library [**?**]. PSBLAS library contains, among others, subroutines for sparse-matrix-by-dense-matrix multiplication, sparse triangular system solution, dot product, sparse matrix and data distribution preprocessing. While addressing a distributed memory execution model the current PSBLAS design does not preclude different implementation pradigms. MLD2P4 (Multi-level Domain Decomposition Parallel Preconditioners Package based on PSBLAS) is a package for parallel algebraic multilevel preconditioners. A purely algebraic approach is used and no information about the underlying physical grid is needed to build the preconditioner. MLD2P4 preconditioner are intended to be used in conjuction with the iterative solvers available in the context of PSBLAS.

Since in MLD2P4 inter-process data communiations is managed through PSBLAS, writing a CAF version give us dual benefit:

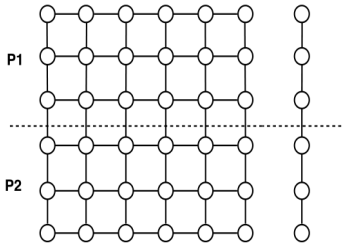
1. First, we can compare MPI and CAF on Krylov subspace methods and simple preconditioners, usign PSBLAS
2. Secondly, we can compare MPI and CAF on multilevel preconditioners, since MLD2P4 data communication is actually managed on PSBLAS.

Additionally, since both PSBLAS and MLD2P4 are open source, once the CAF version is released CAF programmers will be able to use them inside their scientific application.

## 4. Iterative solver Implementation

We converted the PSBLAS code gradually from MPI to CAF, thus having the two communication strategies coexisting in the same code. We detected three communication patterns to modify:

1. The halo exchange
2. Collective subroutines
3. Point-to-point communication in data distribution




Before Halo Exchange

| P1 | | | P2 | | |
|---|---|---|---|---|---|
| loc(i) | glob(i) | val(i) | loc(i) | glob(i) | val(i) |
| 1 | 1 | 1 | 1 | 4 | 2 |
| 2 | 2 | 1 | 2 | 5 | 2 |
| 3 | 3 | 1 | 3 | 6 | 2 |
| 4 | 4 | 4 | 4 | 3 | 3 |

After Halo Exchange

| P1 | | | P2 | | |
|---|---|---|---|---|---|
| loc(i) | glob(i) | val(i) | loc(i) | glob(i) | val(i) |
| 1 | 1 | 1 | 1 | 4 | 2 |
| 2 | 2 | 1 | 2 | 5 | 2 |
| 3 | 3 | 1 | 3 | 6 | 2 |
| 4 | 4 | 2 | 4 | 3 | 1 |

**Figure 1.** Example of halo exchange on 2 images.

In the following, we address the three patterns separately.

### *4.1. Halo Exchange*

The data distribution in PSBLAS [**?**] is best described by referring to an underlying physical problem, such as a PDE discretized on a mesh. Each point in the discretization mesh corresponds to a variable and, consequently, to an index in the coefficient matrix column space; moreover, for each mesh point there is an equation, corresponding to a matrix row. We say that index $i$ depends upon index $j$ whenever $a_{ij} \neq 0$. In the process of distributing data among processes, the original domain is divided into subdomains, with points (and the corresponding equations) in the subdomain assigned to a process. In each subdomain a boundary point is a point depending on points from different domains, which are called halo points. When performing e.g. a matrix-vector product, the values corresponding to halo points will be requested from other processes, with and the amount of exchanged data given by the number of boundary points. This is a so-called halo exchange operation; we can think of halo exchange as a sparse all-to-all communication (`MPI_ALLTOALLV`). It is sparse in the sense that in a good data distribution, each subdomain only interacts with a small subset of all other subdomains. In PSBLAS the halo exchange is by default reimplemented in terms of point-to-point operations; this is because the basic `ALLTOALLV` operation is not commonly optimized for sparse interactions.

The subroutine performing the exchange takes a list of dependencies as input, consisting of variable size blocks, one for each of the communication to perform. Each block contains the remote process with which to exchange data, the number and indices of elements to send, and the number and indices of elements to receive. For the MPI version we can choose between different implementations; the halo_exchange can thus happen through a single call to MPI_ALLTOALLV, by using send and receives communications in synchronized pairs, or it can be split between two separate *send* and *receive* calls. In the latter case the programmer needs to ensure that for each *send* invocation there is a matching *receive*; this is very general, but requires some sort of explicit or implicit buffering of the send data. If the two steps are concatenated in the same call, it is possible to use minimal memory buffer space and maximize performance by employing `mpi_irecv` calls. The exchange then has the following logic flow:

1. Post non-blocking receives;
2. Pack data into buffers and send;
3. Wait for the receive operations to complete;

4. Unpack data from the receive buffers.

For structured grids partitioned along coordinate planes, the halo exchange could be implemented with simple remote coarray accesses. The situation is much more complicated if, as in PSBLAS, we want to handle unstructured meshes and arbitrary domain partitionings. In this case the only way to proceed is to have matched lists of entries to be retrieved and/or sent; a direct access of the remote entries could be implementeed with indirect addressing, with a statement like

```
x(rcv_idxs(1:nrcv(img))) = x(rcv_idxs(1:nrcv(img)))[img]
```

This is perfectly legal, but the implication is that the accesses on the remote image require the indices to travel, and the resulting efficiency would be dubious. The first approach we took then was to have a more "cooperative" handling of data packing between the various processes, with the buffers declared as coarrays.

When rewriting the halo exchange subroutine in CAF, we need to choose if we want to perform the exchange operation using GET or PUT operations. Let's assume that we need to pass a scalar value, stored on the variable *a* from image 1 to image 2. Since communications in CAF are one-sided, this can be done in two ways:

```
if (this_image()==1) a[2] = a
```

or

```
if (this_image()==2) a = a[1]
```

In the first case image 1 performs a PUT operation; in the second case, image 2 performs a GET operation. We have implemented both cases in PSBLAS, but preliminary tests show that the PUT version generally leads to a slightly smaller execution time. Both versions require a coarray buffer. In the GET version it is possible to incorporate the data exchange in the operation of packing the sending data: in this case no other buffer is needed. For the PUT version, it is still necessary to provide a separate receive buffer.

A key factor is the management of allocation of coarrays in the code. Since in CAF allocate and deallocate statements are implicit synchronization barriers, they are significantly more expensive than normal allocations, and therefore they should be avoided as much as possible. Thus we declare communication buffers with the `save` attribute to enable reuse, and we only need to reallocate occasionally when the size of the requested exchange does not fit into the available space. With MPI the buffers are local, and since in most systems allocation is cheap we normally release the buffers to save memor. Before showing performance results for the halo exchange, we make a comparison between MPI and CAF for the halo exchange in terms of impact on software quality. The interface for the halo exchange in psblas is `psi_swapdata`; this interface requires different implementations for the various data types in PSBLAS (integer, real and complex), and for both single vectors and multivectors, i.e. sets of related vectors stored as "tall and skinny" 2D matrices. Any benefits from code simplification are thus accrued multiple times.

Comparing the MPI and the CAF versions we see that there are benefits in :

1. Number of input variables: the MPI version requires 12 input variables, while the CAF version requires 8 input variables

2. Number of lines of code for the single implementation is 185 for MPI and 116 for CAF;

Note that many of the lines of code are actually for testing error conditions and other general tasks in common to the two versions, so the impact on the strict communication part of the code is even larger than appears at first glance.

### *4.2. Collective Subroutines and Point-to-point in Data Distribution*

Here we take into account all the subroutines performing collective communications, in which all images take part in the communication. While rewriting the halo exchange using CAF leads to a reduction in the number of lines in the code, in this case we can expect an increasing in the length of the code. This is because MPI provides an interface for most of the collectives we may need in solving a sparse linear system in parallel. The current language standard defines the following collectives:

1. co_broadcast
2. co_max
3. co_min
4. co_sum
5. co_reduce

Some of the collectives provided by MPI we have investigated reimplementing ourselves; in what follows we are typically emphasizing simplicity over efficiency, a proper study of further implementation techniques will be the subject of future work. In particular we have the implementations for the interfaces caf_scatterv, caf_gatherv, caf_allgatherv, caf_allgatherv caf_gather, caf_alltoallv and caf_alltoall. As an example, the code implementing the caf_alltoall looks like:

```
me = this_image()
np = num_images()
snd_tot=m*np
rcv_start = (me-1)*m +1
rcv_finish = rcv_start + m - 1
do i=1,np
   snd_start = (i-1)*m +1
   snd_finish = snd_start + m - 1
   buffer(rcv_start:rcv_finish)[i]=snd(snd_start:snd_finish)
enddo
!sync to ensure all image has finishe the PUTs
sync all
rcv(1:snd_tot)=buffer(1:snd_tot)
```

The actual code would contain a number of error checking options that are not shown here for clarity. For what it concerns the reduction operations, the `co_reduce` is already part of the standard, and here the difference in code is obvious. For the MPI version we have:

```
call psb_realloc(size(dat),dat_,iinfo)
dat_ = dat
```

**Table 1.** Weak scalability test for a 2D problem ($a_1 = a_2 = \frac{1}{80}$, $b_1 = b_2 = \frac{1}{\sqrt{2}}$, $c = 0$ dim = $idim^2$)

| np | idim | MPI | CAF (*syncimages*) | CAF (events) |
|---|---|---|---|---|
| 1 | 250 | 0.64 | 0.90 | 0.89 |
| 2 | 350 | 0.99 | 1.03 | 1.0 |
| 4 | 500 | 1.37 | 1.58 | 1.33 |
| 8 | 700 | 2.00 | 2.20 | 3.88 |
| 16 | 1000 | 3.03 | 3.82 | 4.41 |
| 32 | 1400 | 5.07 | 5.36 | 6.10 |
| 64 | 2000 | 6.52 | 6.81 | 7.79 |

```
if (iinfo == psb_success_) &
   & call mpi_allreduce(dat_,dat,size(dat),psb_mpi_r_dpk_,&
   & mpi_dnrm2_op,ictxt,info)
```

whereas for the CAF version we have a single statement:

```
call co_reduce(co_dat_,caf_dnrm2)
```

The CAF syntax is more compact and does not differentiate between input and output variables, which suits perfectly our usage and eliminates the need for the extra euxiliary variable dat_.

## 5. Results

CAF and MPI versions of PSBLAS and MLD2P4 have been tested on matrices generated by 2D and 3D PDEs. The experiments were carried out on the Yoda linux cluster, operated by the Naples Branch of the CNR Institute for High-Performace Computing and Networking. Its compute nodes consist of 2 Intel Sandy Bridge E5-2670 8-core processors and 192 GB of RAM, connected via Infiniband. Both PSBLAS, MLD2P4 and OpenCoarrays-1.9 have been compiled on the top of MPICH-3.2 and GNU 7.1.

In PSBLAS and MLD2P4 it is possible to create the matrix corresponding to the PDE $-a_1 \frac{\mathrm{d}}{\mathrm{d}x\mathrm{d}x}(u) - a_2 \frac{\mathrm{d}}{\mathrm{d}y\mathrm{d}y}(u) + b_1 \frac{\mathrm{d}}{\mathrm{d}x}(u) + b_2 \frac{\mathrm{d}}{\mathrm{d}y}(u) + cu = f$ for the 2D problem and $-a_1 \frac{\mathrm{d}}{\mathrm{d}x\mathrm{d}x}(u) - a_2 \frac{\mathrm{d}}{\mathrm{d}y\mathrm{d}y}(u) - a_3 \frac{\mathrm{d}}{\mathrm{d}z\mathrm{d}z}(u) + b_1 \frac{\mathrm{d}}{\mathrm{d}x}(u) + b_2 \frac{\mathrm{d}}{\mathrm{d}y}(u) b_3 \frac{\mathrm{d}}{\mathrm{d}z}(u) + cu = f$ for the 3D problem. The user can set the parameter *idim*. The dimension of the grid is then $idim^2$ for the 2D case and $idim^3$ for the 3D case. The resulting linear system is solved using a Biconjugated Conjugate Gradient Stabilized (BiCGSTAB) with a Block Jacobi preconditioner. The iterations are stopped when the 2-norm of the residuals vector achieved a reduction by a factor of $10^{-9}$

First we tested weak scalability for a 2D problem by keeping (roughly) constant the number of lines per process (Table 1).

For CAF, two versions of the halo exchange have been tested: one performing synchronization through the use of events, the other through the use of the *syncimages* state-

**Table 2.** Strong scaling for a 3D problem: BiCGSTAB with BJAC preconditioner ($a_1 = a_2 = a_3 = \frac{1}{80}$, $b_1 = b_2 = b_3 = \frac{1}{\sqrt{3}}$, $c = 0$,$idim = 200$

| np | MPI | CAF |
|---|---|---|
| 1 | 88.7 | 106 |
| 2 | 46.3 | 46.5 |
| 4 | 34.9 | 23.3 |
| 8 | 1.72 | 1.45 |
| 16 | 7.31 | 9.94 |
| 32 | 5.71 | 5.12 |
| 64 | 4.04 | 3.61 |

**Table 3.** Strong scaling for a 3D problem: BiCGSTAB with with a 2 level preconditioner ($a_1 = a_2 = a_3 = 1$, $b_1 = b_2 = b_3 = 0$, $c = 0$,$idim = 200$

| | MPI | | CAF | |
|---|---|---|---|---|
| np | tprec | tsolve | tprec | tsolve |
| 1 | 37.5 | 35.8 | 37.4 | 35.6 |
| 2 | 19.9 | 27.1 | 19.8 | 26.9 |
| 4 | 10.3 | 15.5 | 10.5 | 15.3 |
| 8 | 5.56 | 8.46 | 5.37 | 9.87 |
| 16 | 3.71 | 5.64 | 2.35 | 5.87 |
| 32 | 2.19 | 3.68 | 2.44 | 3.72 |
| 64 | 1.45 | 4.0 | 2.12 | 3.72 |

ment. While we expected synchronization through events to perform better, we actually observed that the version with *syncimages* is faster especially on many processes. In this case, MPI and CAF versions have total execution times that are very similar. Secondly, strong scalability tests have been performed for both 2D and 3D problems. Again, halo exchange with sync images performs better: for this reason in the following we show only results for this case. Table 2 shows results for the 3D problem.

Finally, we show results for the 3D problem, using the multilevel preconditioner implemented in MLD2P4. BiCGSTAB is used, with a multiplicative multilevel preconditioner, smoothed decoupled aggregation algorithm. The iterations are stopped when the 2-norm of the residuals vector achieved a reduction by a factor of $10^{-6}$ and a Jacobi smoother is used. The coarsest level matrix is distributed among processes and an incomplete LU factorization is used to solve the coarsest level matrix. Results are shown in Table 3. Again performances of MPI and CAF are comparable in term of execution time. In the table, both the preconditioner building time (tprec) and the solution time are shown (tsolve).

## 6. Conclusions

We successfully obtained a first migration of the PSBLAS library from MPI to CAF. Since MLD2P4 communication is based on the PSBLAS framework, most of MLD2P4 preconditioners can exploit underlying CAF communication with little effort. We point

out that in the migration from MPI to CAF we retained as much as the original code and structure that we could. While this can lead to CAF code which is not as clear and readable as it would be if the original software was thought and written in CAF from the beginning, we think that the previous examples show the capability of CAF to express parallelism in a simpler way than MPI. In all cases, CAF code is shorter, clearer and easier to read. However, the CAF programmer need to provide an interface for most of MPI collective subroutines, which can increase the lenght of the code.

We tested both the MPI and CAF versions on problems arising from discretization of 2D and 3D PDEs. Neither CAF nor MPI version has shown a clear superiority in terms of performance, with execution time almost identical in the two cases. The initial migration from MPI to CAF was achieved without a loss of performance; however this is only the first step. In future work we plan to explore in detail possible alternative strategies for remote indirect addressing, which will likely include close interaction with the development of the compiler and of the OpenCoarrays transport layer. Additionally, we provided few examples of how CAF programmer choices can affect performance (for example: ”get” versus ”put” for one-sided communication, events versus *syncimages* for synchronization, clever coarrays allocation).

## 7. Acknowledgments

This work has been partially funded from the European Union’s Horizon 2020 Research and Innovation Programme under Grant Agreement No. 676629 (Project EoCoE). The authors wish to thank Dr. Pasqua D’Ambra (CNR, Napoli, Italy), for her help in running the experimental tests.