Solving Sparse Linear Systems of Equations using CAF

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

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

Abstract. Fortran Coarrays (CAF) has been part of the Fortran standard since Fortran 2008 and it provides a syntactic extension of Fortran to support parallel programming. 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 the best-usage strategies implemented in PSBLAS and MLD2P4 and we show some performance results obtained on linear systems arising from discretization of 2D and 3D PDEs.

Keywords.

1. Introduction

Fortran has been and still is a dominant language in High Performance Computing. In past decades, Fortran has greatly developed, and now supports class abstraction, object-oriented programming, pure functions, and coarrays [1]. Fortran Coarrays (CAF) has been part of the Fortran standard since Fortran 2008 and it provides a syntactic extension of Fortran to support parallel programming. Following Partitioned Global Address Space programming model, with CAF the program is replicated among a certain number of images, executing asynchronously. Combining, elegancy and simplicity, CAF allows easily for one-sided communications, potentially reducing synchronization and code complexity. CAF is a language-based approach, which means it is bounds on specific compilers: when a good implementation is available, compiler can concurrently consider serial aspects and communication for optimization. Additionally, if vendors compiler provides support for OpenAcc, the user can easily get a dual benefit form the combined use of CAF and GPUs. MPI (Message Passing Interface) defines a suite of functions for communication exchange between processes. Differently from CAF, it is library-based: it means that while independent from a specific compiler, the user is bound to the vendor's fine-tuning of the MPI library [2]. Although MPI is the de facto standard for parallel programs running on distributed memory systems and few scientific softwares are written in CAF, we think that many scientific applications could benefit from the use of CAF. For this to be possible, studies have to be carried on the impact of CAF on performances and quality of scientific softwares. In this paper we focus on the solution of large and sparse linear systems of equations and we propose the migration of the PSBLAS [3] and MLD2P4 [4] 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 [5] (in which early versions of CAF and MPI have been used). In [2] 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 [6], both opensource and easily accessible.

2. Fortran Coarray

Coarray Fortran extends 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 PGAS model threads share the global address space, each having a portion of space which is local to it. Besides CAF, Unified Parallel C (UPC) [7], Chapel [8], X10 [9] and SHMEM are all based on the PGAS model. With CAF, Fortran users do not need to invoke subroutines from an external API (as in MPI), nor preprocessing directives (as in OpenMP) to create parallel programs. Since most of the parallel bookkeeping is handle behind the scene by the compiler, CAF achieves a good trade-off between readability and performance and 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 passing a non-contiguous array using MPI or to the fact that with CAF there is no need to write separate code for sending and receiving a message. Coarrays are fully supported by Cray and Intel compilers. Although GNU compilers provide only partial coarray support (only single image), the open source openCoarrays project provides two coarray transport layers based on MPI and on GASNet [6].

In CAF the program is replicated a fixed number of times and each replication (called images) executes asynchronously having its own set of data objects. Some variables can be declared as coarrays through the use of coindices (indicated by square brackets) with a syntax very similar to normal Fortran array syntax.

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

Coarrays can be accessed by the other images, thus enabling data exchange.

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

Each image is assigned a unique index, through which the programmer can control the flow of the execution. Image indices start from 1 (not from 0 as MPI process ranks) and can be retrieved using the intrinsic function *this_image*(). The number of total images can be retrieved using the function *num_images*(). The total number of images can be different from the total number od processors and it can be decided at compile, link or runtime depending on the implementation.

CAF communications are synchronous and non-blocking. To avoid 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 tohers) while sync images allows for explicit synchronization of single images. Moreover, with CAF events, an image can use an event post statement to notify another images or it can wait for an event posted by another image before continuing execution. This allows for more sophisticated synchronization strategies.

Other than explicit synchronization, in a CAF program there can be implicit synchronization points. The programmer must be aware of this, because they can lead to a drastic change in performance. An example of an implicit synchronization point is allocation or deallocation of an allocatable coarray:

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

Of course, for code between synchronizations, the compiler can use the usual optimization techniques as if only one image is present [10].

3. PSBLAS and MLD2P4

Most of 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 usually ineffective and iterative solvers are preferred. 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 [11]. Among iterative methods, Krylov methods are based on minimizing the residual over a Krylov subspace and are considered among the most important [12]. The Conjugate Gradient (CG) [13], the Generalized Minimal Residual (GMRES) [14] and the Biconjugate Gradient (BiCG) [15] are all Krylov methods. 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 [16]. Multigrid can be used as stand-alone solvers [17] or as preconditioner for a Kylov subspace method [18]. 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 the serial sparse BLAS [19], while the inter-process massage exchanges are encapsulated in an application layer inspired by the Basic Linear Algebra Communication Subroutines (BLACS) library [20]. PSBLAS library contains, among the 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. Implementation

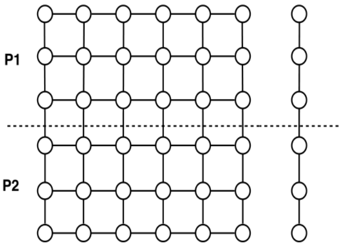
We converted the PSBLAS code gradually from MPI to CAF, thus having the two communication strategies coexisting in the same code. We detected three communication patterns to modify:

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

In the following, we address the three of them separately.

4.1. Halo Exchange

In PSBLAS [3], data distribution is performed keeping in mind the underlying physical problem, such as the discretization mesh of the PDE. Each point in the discretization mesh corresponds to a variable and, consequently, to an index in the coefficient matrix. We say that index i depends upon index j if $a_{ij} \neq 0$. In the process of distributing data among processes, the original domain is divided into subdomains. In each subdomain a boundary point is a point depending on points from different domains. An halo point is the one for which there is at least one boundary point in another domain depending upon it. This means that when performing matrix and vector operations (such as a matrix-vector production), halo points will be requested by other processes and the amount of exchanged data is given by the cardinality of the boundary points. Every time this happens we perform a so-called halo exchange operation. We can think of halo exchange as a sparse all-to-all communication (or as an MPI_ALLTOALLV operation, in MPI jargon). In order to obtain better performances in PSBLAS halo exchange is actually reimple-



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.

mented in terms of point-to-point operations. The subroutine performing the exchange takes a list of dependencies as input. The list consists of variable size blocks, one for each of the communication to perform. The list terminates with -1 and each block consist of several elements:

1. The first entry identifies the process to which exchange data.
2. The number of elements to receive
3. The local indices of the elements to receive
4. The number of elements to send
5. The local elements to send

The order of the blocks, determining the order in which communications take place, is defined through an internal algorithm. For the MPI version we can choose between different strategies by setting a suitable flag as an input variable. The halo_exchange can thus happen through a call to MPI_ALLTOALLV, by using send and receives communications in synchronized pairs, by performing only a send or a receive (in this case, the code needs to complete the communication by calling the same subroutine with the complementary communication in a later stage) or by using mpi_irecv and mpi_send. The latter choice is the standard one, since it leads to the best results in terms of performance. In this case data exchange actually takes place through the use of two buffers (snd_buf and rcv_buf) according to the following flow:

1. pack data into the snd_buffer
2. call the non-blocking mpi_irecv using rcv_buffer as output
3. complete the exchange calling blocking mpi_snd using snd_buffer as input
4. upack data from the rcv_buffer

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 implemented both cases in PSBLAS, but preliminary tests show that PUT 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 instead, the rcv_buf is still present. A key factor is the management of allocation of coarrays in the code. Since in CAF allocate and deallocate statements are implicit synchronization barriers, for the sake of performance they should be avoided unless strictly necessary. For this reason in PSBLAS code we declared the coarray buffer with the save attribute and we reallocate it only when the actual amount of data to exchange does not fit the current size:

```
if (psb_size(buffer) < xchg%max_buffer_size) then
  if (allocated(buffer)) deallocate(buffer)
  allocate(buffer(xchg%max_buffer_size)[*], stat=info)
  if (allocated(sndbuf)) deallocate(sndbuf)
  if (info == 0) allocate(sndbuf(xchg%max_buffer_size), stat=info)
  if (info /= 0) then
    info = psb_err_internal_error_
    call psb_errpush(info ,name, a_err='Coarray␣buffer␣allocation')
    goto 9999
  end if
end if
```

Note that with MPI, the opposite is true: we deallocate the buffers as soon as we can, in order to save run-time memory. Before performance results are shown for the halo exchange, we can 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 5 implementations for each of the available datatypes in PSBLAS (integer, real, double precision, complex and double precision complex). Each of them needs to be implemented for the the real vector, the real matrix and the two derived types defined in PSBLAS psb_X_vect and psb_X_multivect. A second interface, psi_swaptran performs a transpose halo exchange. As a consequence, we need 40 implementations of the halo exchange: these are all very similar with some minor modifications. It is clear that any impact on the code, although relatively small on the single implementation, has to be multiply by 40. Keeping this in mind, we detected two differences between the MPI and the CAF version:

1. number of input variables: the MPI version requires 12 input variables, while the CAF version requires 8 input variables
2. the number of lines of code for the single implementation is 185 for MPI and 116 for CAF

Less input parameters results in a more clear interface, and we get a significant reduction in the number of lines of the code (regarding only the halo exchange).

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. For CAF collective subroutines are reduced in number; they are:

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

They involve synchronization within them, but not at the start or at the end. With CAF, we need to write from scratch some of the collective subroutines we need. We do this in the psb_caf_mod. In this module, 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 for the caf_alltoall has been provided.

```
subroutine caf_alltoall(snd,rcv, m, info)
  implicit none
  integer(psb_ipk_), intent(in) :: m
  integer(psb_ipk_), intent(in) :: snd(:)
  integer(psb_ipk_), intent(out):: rcv(:), info
  !Local
  integer(psb_ipk_) :: me, np,i, snd_start, snd_finish, snd_tot
  integer(psb_ipk_), allocatable :: buffer(:)[:], rcv_start, rcv_finish
  if ( m < 0) then
    print*,'Error, m must be greater or equal to zero'
    info = -1
  endif
  me = this_image()
  np = num_images()
  snd_tot=m*np
  if (allocated(buffer)) deallocate(buffer)
  allocate(buffer(snd_tot)[*], STAT = info)
  if (info /= 0) then
     print*,'allocation error', info
     return
  endif
  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
    if (rcv_finish > snd_tot) then
      print*,'Error, rcv_finish > snd_tot'
      info = -2
      return
    endif
```

```
      if (snd_finish > size(snd,1)) then
        print*,'Error,␣snd_finish␣>␣size(snd,1)'
        info = -3
      endif
      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)
    if (allocated(buffer)) deallocate(buffer)
  end subroutine caf_alltoall
```

For what it concerns the reduce operations, we show a comparison between CAF and MPI based on the code for the reduce operation based on the norm2. First, the code for the MPI version:

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

And the code the CAF version:

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

In this case the CAF syntax is more compact and there is no need to use the extra euxiliary variable dat_. Finally, we modified the psb_matdist subroutines, an utility subroutine that uses point-to-point communications to distribute a matrix among processors according user defined data distribution. In this case, for both MPI and CAF we are using one-sided communications. In the case of CAF, to ensure performance optimizations, we used events to ensure synchronization among images.

5. Conclusions

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.

References

[1] Michael Metcalf, John Reid, and Malcolm Cohen. *Modern Fortran Explained*. Oxford University Press, Inc., New York, NY, USA, 4th edition, 2011.

[2] S. Garain, D. S. Balsara, and J. Reid. Comparing Coarray Fortran (CAF) with MPI for several structured mesh PDE applications. *J. Comput. Phys.*, 297(C):237–253, September 2015.

[3] S. Filippone and M. Colajanni. PSBLAS: a library for parallel linear algebra computations on sparse matrices. *ACM Trans. Math. Softw.*, 26(4):527–550, December 2000.

[4] P. D'Ambra, D. di Serafino, and S. Filippone. MLD2P4: a package of parallel algebraic multilevel domain decomposition preconditioners in Fortran 95. *ACM Trans. Math. Softw.*, 37(3), 2010.

[5] R. W. Numrich, J. Reid, and K. Kim. *Writing a multigrid solver using Co-array Fortran*, pages 390–399. PARA '98. Springer Berlin Heidelberg, 1998.

[6] A. Fanfarillo, T. Burnus, V. Cardellini, S. Filippone, D. Nagle, and D. Rouson. OpenCoarrays: Open-source transport layers supporting coarray fortran compilers. In *Proc. of 8th Int'l Conf. on Partitioned Global Address Space Programming Models*, PGAS '14, pages 4:1–4:11. ACM, 2014.

[7] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[8] B.L. Chamberlain. Chapel. In *Programming Models for Parallel Computing*, chapter 6, pages 129–159. MIT Press, 2015.

[9] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[10] R. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998.

[11] Richard Barrett, Michael Berry, Tony F Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, 1994.

[12] Ilse CF Ipsen and Carl D Meyer. The idea behind krylov methods. *American Mathematical Monthly*, pages 889–899, 1998.

[13] Jonathan Richard Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[14] Youcef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[15] Henk A Van der Vorst. Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing*, 13(2):631–644, 1992.

[16] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003.

[17] Achi Brandt. Multi-level adaptive solutions to boundary-value problems. *Mathematics of computation*, 31(138):333–390, 1977.

[18] James H Bramble, Joseph E Pasciak, and Jinchao Xu. Parallel multilevel preconditioners. *Mathematics of Computation*, 55(191):1–22, 1990.

[19] Iain S Duff, Michael A Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the blas technical forum. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):239–267, 2002.

[20] R Clint Whaley and J Dongarra. A user's guide to the blacs v1. 1. *Computer Science Departement, University of Tennessee Knoxville*, 5, 1997.