Gabriel 1.3.2: A Fortran library for scalable, verified and convenient message passing

John Donners*  
*SURFsara, Amsterdam

Abstract

The Gabriel library combines features in MPI-3.0 and Fortran 2015 with classic advantages of Fortran to improve the use of MPI by geophysical models on regular grids. The user can define a composition of arrays with one collective call over an MPI communicator. Another call then constructs a distribution of messages, e.g., to exchange halo regions. This greatly simplifies the coding of halo exchanges, since the developer never has to define the halos explicitly. It allows flexible decompositions and variable halo widths without additional code. This flexibility has been used in the LES-COAST model, a large eddy simulation, to change a one-dimensional decomposition into a two-dimensional decomposition. This resulted in a maximum rank count about ten times higher for the application. In the NEMO model, replacement of hand-coded halo exchanges resulted in a lower overall model performance. The performance of the Gabriel library depends on the implementation of MPI derived datatypes in the underlying MPI-library, which is less than expected. In conclusion, Gabriel offers scalability, verification and ease of use for models on regular grids.

1. Introduction

Today’s supercomputers necessitate the use of distributed computing to reach their full potential. The Message Passing Interface ("MPI") is the standard that is used by most applications to implement distributed computing. MPI was first defined in 1994 [1] and has been enhanced with new functionality with the release of MPI-2.0 in 2003 [2]. It took until 2012 before further, significant enhancements were added with the release of MPI-3.0 [3]. At the moment of writing, all major developers of MPI implementations have released MPI libraries that support nearly all of the MPI-3.0 functionality. MPI-3.0 offers over 400 subroutines to communicate information in a standardized and fast way between distributed processes.

The Fortran 90 programming language [4] is adopted by many established earth science models. Because its functionality was geared towards scientific computing from the start, it provides a good performance, rich functionality and hides the details of the computer architecture from the user. Recent standardization efforts [5] have added more modern programming paradigms, including information hiding and object-oriented programming, while still being compatible with most of the earliest standards. With these features, applications can reach a higher level of abstraction with incremental changes. Also the development of reusable libraries is simplified. Many compiler vendors have implemented a large part of the latest Fortran standards in recent releases. The latest developments of the Fortran language [6] were actually incorporated to comply with the MPI-3.0 standard and will be included in Fortran 2015.

The use of advanced MPI features like derived types is hampered by a few issues: although the performance of derived types was studied and optimized [7, 8], the performance has been found lower than expected, while the other issue is that debugging becomes harder: a segmentation fault in an MPI call can be caused by the incorrect
definition of the MPI derived type, but the definition and application of derived types is usually separated in the code.

The main aim of the Gabriel library is to act as a simple MPI layer for models. At the moment, it is geared towards models defined on regular grids. The main innovations of Gabriel are:

- The user defines the composition of the arrays from all ranks.
- Each rank provides the region of the array that is computed locally and how that fits in the global domain.
- The user then asks for a distribution for halo exchanges of one composition or a transformation from one composition to another. The library automatically deduces neighboring ranks and the subarrays to communicate.
- The user supplies array indices as defined in his application.

The library is in the public domain and can be found at https://github.com/jdonners/gabriel.

2. Global overview

The Gabriel library provides just a few derived types and a small set of auxiliary routines to the user. In line with modern programming techniques, the derived types are opaque and the type can only be accessed by its type-bound procedures. This decouples the library environment from the user environment, which is crucial to improve the reuse of software. Gabriel does not copy any data in or out of buffers, it only defines MPI derived datatypes and calls MPI communication routines. Its overhead in terms of CPU cycles or memory is therefore negligible. Consequently, the performance depends on the underlying MPI library and its implementation of handling derived datatypes and neighborhood collectives.

Firstly, the user needs to define the composition of the data structure at all MPI ranks in the communicator. The Gabriel library provides the BOX type for regular grids, which can be used to describe a composition of arrays of any rank (i.e., the number of dimensions). This is a collective MPI call.

The DISTRIBUTION type describes a communication pattern between all ranks in the MPI communicator. It holds the lists of messages that need to be sent and received, which includes the rank and derived type for each message. Type-bound procedures automatically generate distributions based on the input composition. These subroutines deduce the neighboring ranks and create the MPI derived datatypes for all messages. These are also collective MPI calls.

At the moment, Gabriel includes automatic distributions for

- Halo regions: set up MPI communications for all halo regions of the array, i.e., parts that are not computed locally.
- Transformations: setup MPI communications to transform one composition into the other. This is useful to change e.g., a row-decomposed domain to a column-decomposed domain, or to change from one-dimensional to a two-dimensional decomposition.

The HALO type is the basic building block that contains an MPI derived datatype and its description. The HALO type can be a chain of types, e.g., two different subarrays of a single array, or the same subarray of multiple arrays. The description of the MPI derived datatype is used to check if the input array matched the description before the actual MPI communication. If the HALO type does not match the input array, the library gives an error including the expected and actual array bounds, whereas the user would otherwise be confronted with a segmentation fault deep inside the MPI-library. This check can be disabled at runtime, to minimize overhead.

At the moment, there are three different types of HALO:

- Subarray: a subarray of a larger array.
- Joined: the same HALO of different variables. A common case is to join the same subarray of different variables.
- Combined: a combination of different HALOs. A common case is to combine different subarrays of the same variable.
Note that the HALO type only needs to be used explicitly by the user if a DISTRIBUTION type should be fine-tuned with additional communications.

In line with common Fortran usage, errors in Gabriel are by default fatal. If the user supplies an optional err argument, Gabriel returns the error code and the application can recover. Note that errors in MPI calls are by default fatal, independent of the optional err argument. The user can change the MPI error handler to recover from MPI errors (if possible).

3. Example

Let us assume that we want to study a 1D model that is discretized in a 1D array and distributed across 2 processes:

<table>
<thead>
<tr>
<th>MPI rank</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indices of array A</td>
<td>0123456</td>
<td>3456789</td>
</tr>
<tr>
<td></td>
<td>......** **....</td>
<td></td>
</tr>
<tr>
<td></td>
<td>..=Locally computed **=Halo region</td>
<td></td>
</tr>
</tbody>
</table>

The array A consists of 7 elements in total, of which 2 elements are the halo region. To calculate a numerical derivative on the locally computed domain, each MPI rank needs the values from its neighbor in the halo regions. The code in Gabriel to communicate the halo regions, is

```fortran
program one_two
  use mpi
  use gabriel
  implicit none
  real,dimension(:),allocatable :: A
  type(box) :: a_comp
  type(distribution) :: dist
  integer ierr,rank
  call MPI_Init(ierr)
  call MPI_Comm_rank(MPI_COMM_WORLD,rank,ierr)
  call gabriel_init()
  if (rank.eq.0) then
    allocate(A(0:6))
    A=0.0
    call a_comp%initialize(A,(/0/),(/4/),MPI_COMM_WORLD)
  endif
  if (rank.eq.1) then
    allocate(A(3:9))
    A=1.0
    call a_comp%initialize(A,(/5/),(/9/),MPI_COMM_WORLD)
  endif
  call dist%halo(a_comp)
  call dist%create
  call dist%update(A)
  write(*,'(a,i2,a,7f5.1)')'Rank=',rank,' A=',A
  call MPI_Finalize(ierr)
end
```

The Gabriel library interface is defined in a fortran module (line 3). The gabriel_init call reads the GABRIEL_VERBOSE environment variable (line 12). The calls at line 12 and 17 are used to define the BOX composition of array A at the two MPI processes as described at the start of this section. The call dist%halo at line 25 creates the MPI types to update halo regions of arrays with the composition a_comp. The call dist%create
at line 26 creates the MPI graph communicator and the call at line 27 updates the halo regions of A. All calls at lines 12, 17 and 25 to 27 are collective MPI calls.

and the output of this program is

```
Rank= 0 A=  0.0  0.0  0.0  0.0  0.0  1.0  1.0
Rank= 1 A=  0.0  0.0  1.0  1.0  1.0  1.0  1.0
```

Note that the user does not need to explicitly define the halo region. The halo region could have been larger or smaller, or the halo sizes could be different on different ranks. In all these cases, the Gabriel calls would be the same. For multi-dimensional arrays the number of calls stays the same, only the arguments with the array indices would expand.

### 4. Features

Figure 1 shows a regular, 2D domain decomposition with the size of the full array, including the overlapping halo regions, for the central rank. The computational domain are the gridpoints that are calculated locally. The box composition routine checks that none of the computational domains across all ranks in the communicator overlap. Then halo distribution routine searches for overlap between the local computational domain and the full array size of each rank. For each overlapping region, it creates a derived type and adds a rank to send data to. Next, it searches for overlap between other computational domains and the full size of the local array. For each overlapping region, it creates a derived type and adds a rank to receive from. Note that Gabriel also communicates the halo regions at the corners of the array for ranks that are diagonally adjacent.

![Regular domain decomposition supported by Gabriel](image)

Figure 1 Regular domain decomposition that is supported by Gabriel. The local computational domain is green and neighboring computational domains are light brown, halo regions are shaded. The local array is bounded by the thick black line.

Gabriel neither assumes any neighboring MPI ranks, nor assumes sizes or shapes of halos. As a consequence, besides regular domain decompositions, it naturally supports several optimization strategies used regularly in geophysical models as well:

- Removing ranks that would represent holes in the domain, e.g., land masses in an ocean model, can provide significant savings. For a global ocean model, the land mass represents 30% of the total surface.
- Some numerical schemes allow a trade-off between halo size and communication count, e.g., successive overrelaxation.
- The global domain could be decomposed along one direction to minimize the number of neighboring ranks, or along multiple directions to minimize communication size.
- Irregular domain decomposition could alleviate poor load balance due to different active processes on different domains.
In the first two cases, the calls to the Gabriel library would be unchanged, both to setup and to communicate. The user only needs to supply the full array and the bounds of the computational domain. In the latter two cases, the user would only need to modify the bounds of the computational domain accordingly at setup. Figure 2 shows an example that can be supported by Gabriel.

![Figure 2 Irregular domain decomposition with periodic boundaries, holes in the domain and a wider halo on the right side. The local array is bounded by the thick black line. The local computational domain is green and neighboring computational domains are light brown; halo regions are dark brown and the blue halo regions at the top are filled as periodic boundary by Gabriel. The blue halo region in the center of the domain is not covered by any local, computational domain and will therefore be ignored by Gabriel.]

Fortran recently supports assumed-rank arrays, i.e., dummy arguments for arrays with variable rank. This feature allows the use of arrays of any rank with Gabriel.

The Gabriel library supports periodic boundary conditions with an optional logical array periodic. Periodicity can be defined independently for each dimension of the array. It automatically detects the global domain size and adds communications for the halos along the edges of the global domain. Note that this includes the corners and ribs in three-dimensional and higher-dimensional arrays.

The Gabriel library utilises the property of Fortran arrays that the user can define both its lower and upper bounds. This often blends in nicely with the use of indices in the existing code. The global ordering of arrays is derived from the ordering of the local array indices on each rank. However, many models use different indexing, e.g., the local array on each rank starts at index -1. In this case, the optional argument offset can be used to provide the indices for the global view of the array. The offset is added to the lower and upper bounds of the local input array.

The Gabriel library also correctly executes when the application is run as a single process without any modification, which simplifies debugging of the application.

The library provides functions to automatically setup communication patterns for halo exchanges and transformations. The user can manually add further HALOs to the resulting DISTRIBUTION.

### 4. Applications

This section describes two applications of the Gabriel library. Firstly, the LES-COAST [9] model was changed partially to a two-dimensional decomposition, resulting in a higher scalability. Secondly, the most-used halo communication routines in the well-known NEMO ocean model [10] have been replaced with a few calls to the Gabriel library to check its performance.
4.1. LES-COAST

The library primarily originates from the parallelisation of the LES-COAST model [9] during a PRACE preparatory access project. This model was already parallelised using MPI and uses the multi-grid method to solve the pressure equation. One of the main ideas of the Gabriel library, the use of Fortran array indices to infer the composition of all processes, stems from this model. The scalability of the model was low and was furthermore limited by the domain decomposition along only one direction. The coarsening of the multi-grid method complicates the further parallelization of the model, as the number of processes becomes quickly higher than the number of gridpoints at the coarsest resolution along the decomposed dimension.

The decomposition along one dimension therefore had to be changed into a decomposition along two dimensions to increase the maximum processor count. The number of neighbors in a regular domain decomposition then increases from two to eight, complicating the MPI-communication pattern for the halo regions. The periodic boundary conditions could be mainly implemented as a simple array copy in the original model, but require MPI communication in the two-dimensional decomposition.

Only the decomposition of the multi-grid method was changed and therefore the pressure data has to be transformed between the two decompositions at the start and end of the call to the multi-grid method. The Gabriel library is also used for this transformation and the transformation of all other input arrays.

The figure below shows the speedup of the code with the Gabriel library, relative to the maximum number of processors that could be used with the original code. The used benchmark is the 3D simulation of a fluid tank with 512x1024 grid points in the horizontal and 64 grid points in the vertical. Although the original code is slightly faster at 32 cores, the code using the Gabriel library can scale to 512 cores and results in a 10-fold increase in performance. The Gabriel library could hide the complex and error-prone MPI communication pattern from the developer and improved the scalability of the LES-COAST model from at most tens of MPI tasks to hundreds of MPI tasks.

![Speedup Graph](image)

4.2. NEMO

The NEMO ocean model [10] has been adopted by the PRACE Unified European Applications Benchmark Suite (UEABS) as one of twelve scalable, relevant and publicly available codes. The model uses a regular, two-dimensional domain decomposition.

The NEMO model uses temporary arrays to hold the halo data that is sent to and received from its neighbors. Each exchange requires an additional copy before sending and after receiving the data.

A NEMO task first exchanges data with its east-west neighbors before it exchanges data with its north-south neighbors. The main part of the halo exchange subroutines could be replaced with a few calls to the Gabriel library:

```
! Initialization
from=(/1+jpreci,1+jprecj/)
to=(/ihom+jpreci,ijhom+jprecj/)
call ptabbbox%init(pt2d,from,to,mpi_comm_apa,offset=(/nimpp,njmmm/))
call ptabdist%halo(ptabbbox)
call ptabdist%create
...
! Halo update
call ptabdist%update(pt2d)
```
Table 1 shows the performance of the UEABS NEMO GYRE benchmark for two configurations. The Gabriel library does not use temporary arrays and exchanges data with all neighbors in one step, which might suggest that the overall exchange can be done faster. However, the results clearly show that the performance of the Gabriel library is lower than or at best equal to the existing implementation in NEMO. The difference between the original and the Gabriel version for the Intel MPI library is large, while it is small for the HPC-X library (based on OpenMPI, optimized for Mellanox hardware). Benchmarks with MPICH 3.2 show results very similar to Intel MPI. The last line shows a larger benchmark running with 1536 MPI tasks, which shows that the scalability of the Gabriel library is on par with hand-coded sends and receives. Note that these runs have not been analysed in detail and the results only indicate the impact of Gabriel for different MPI libraries. These results should not be used as a performance comparison of the MPI-libraries.

5. Discussion
As mentioned before and as shown in the NEMO benchmark, the performance depends a lot on the underlying MPI library. The best performance is achieved with the original NEMO implementation, although the performance of the Gabriel library is on par for some MPI libraries. The overhead of the MPI libraries for handling derived datatypes, i.e., to communicate non-contiguous memory blocks, is high [11] and hand-coded loops to copy data to and from contiguous communication buffers are fastest at this moment.

Although a lot of research has gone into handling MPI derived datatypes, more is needed to show its performance benefits. Recent hardware developments for the network, like direct access to non-contiguous memory regions, are expected to be included and optimized in future MPI library releases and hopefully result in improvements in their performance of derived datatypes.

The use of the latest Fortran and MPI features did uncover some bugs in the compilers and libraries. Therefore, the current release only supports allocatable arrays of either single- or double-precision REALs. Some of the bugs have already been resolved in recent releases. E.g., Intel Fortran 17.0.0 can already support static, automatic or pointer arrays. Another error was uncovered in a low-level library to optimize communications on specific hardware. Gabriel therefore requires recent compilers and MPI libraries to work properly. The included configure script and test programs check some of these and report back incompatibilities or bugs. The README mentions the minimal versions for popular Fortran compilers and MPI libraries.

Gabriel uses one MPI_Neighbor_alltoallw call for to update a DISTRIBUTION and therefore it is straightforward to implement a non-blocking variant. This is planned for a future release. Again, the performance will depend on the underlying MPI library. However, this could be a different benefit of Gabriel for application developers, since it would be difficult to implement the hand-coded halo exchanges as asynchronous threads. It would be easier to overlap communication with computation.

The same principle to define a composition of arrays across all MPI ranks can also be used for other purposes. One example is to write such an array to disk using parallel MPI-IO. Another example is to transfer the data for (parallel) in-situ visualization. This has not yet been planned for a future release.

6. Conclusions
The Gabriel library provides a convenient yet powerful way to setup MPI communications for models on regular grids. It supports both regular and irregular domain decompositions. The user creates a composition to describe
the connection between arrays on different ranks. The Gabriel library can automatically create distributions from a composition, e.g., to update halo regions, without the user having to define neighboring ranks, halo sizes or halo locations. Gabriel simplifies the required coding to implement MPI communication and therefore provides the opportunity to use more complex and more scalable domain decompositions. The performance of Gabriel depends on the underlying MPI library, since Gabriel has virtually no overhead of its own when communicating. Unfortunately, current MPI libraries show a negative performance impact or little benefit at best. The scalability of the Gabriel library is on par with manual implementations of a halo exchange. Future plans include non-blocking communications and possibly MPI-IO and transfer to on-line analysis software.

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EU’s Horizon 2020 research and innovation programme (2014-2020) under grant agreement 653838 and the Dutch National Organisation for Research (NWO).

References