Horizon 2020
FET-Proactive - towards exascale high performance computing

Programming Model INTERoperability ToWards Exascale
671602

Best Practice Guide for Writing GASPI - MPI Interoperable Programs

Tiberiu Rotaru

June 10, 2016, version 1.0

Abstract

The aim of the best practice guide is to sketch common practices related to the MPI - GPI-2 interoperability. The present document tries to show via concrete examples different modalities of mixing MPI and GPI-2 code within the same parallel program, pointing to some common practices that should be taken into account when proceeding with this.

1 Introduction

The Message Passing Interface (MPI) has been considered the de facto standard for writing parallel programs for clusters of computers for more than two decades already. Although the API has become very powerful and rich, having passed through several major revisions, new alternative models that are taking into account modern hardware architectures have evolved in parallel. Such a model is the Global Address Space Programming Interface (GASPI), GPI-2 representing an implementation of the GASPI standard.

GASPI is a modern specification of a compact API for the development of parallel applications. It *aims at initiating a paradigm shift from bulk-synchronous two-sided communication patterns towards an asynchronous communication and execution model.* GPI-2 is an open source implementation of the GASPI specification and it is freely available at www.gpi-site.com/gpi2 and www.github.com/cc-hpc-itwm/GPI-2.

The GASPI standard promotes the use of one-sided communication, where one side, the initiator, has all the relevant information (what, where from, where to, how much, etc.) for performing the data movement. The benefit of this is decoupling the data movement from the process synchronization. It enables the processors to put or get data from remote memory, without engaging the corresponding remote processor, having no synchronization point for every communication request. However, some form of synchronization is still needed in order to allow the remote process get notified upon the completion of an operation.

GASPI provides so-called weak synchronization primitives which update a notification on the remote side. The notification semantics is complemented with routines that wait for updating a single or a set of notifications. GASPI allows for a thread-safe handling of notifications, providing an atomic function for resetting a local notification with a given ID (returns the notification value before reset). The notification procedures are one-sided and only involve the local process.

2 Interoperability

GASPI aims at providing interoperability with MPI in order to allow for incremental porting of applications. GPI-2 supports this interoperability with MPI in a so-called mixed-mode, where the MPI and GASPI interfaces can be mixed. In the present document we aim at providing useful hints, via concrete examples, on how this interoperability with MPI is allowed by GPI-2.

MPI-based programs that are well structured and algorithmically clear can be ported with reasonable effort to GPI-2, as shown in [1]. However, entirely porting a large MPI application to GPI-2 might become a challenging task, especially

when dealing with complex legacy codes, in the absence of a good understanding of the application's logic. In such cases, it may be more adequate to proceed incrementally, by firstly identifying distinct MPI communication patterns that can be replaced gradually by GPI-2 communication. The application developers do not need to port everything, especially when using some external MPI code embedded in libraries. They may start replacing some critical parts of the application, where GPI-2 is known to perform better than MPI or it can take advantage of features like one-sided communication, weak synchronization and thread-safety.

As the interoperability between different programming models is in general a complex theme that may assume a more sophisticated design, consisting of several layers above MPI, GPI-2 or other communication libraries (`http://www.intertwine-project.eu/`), we highlight in this document only the aspects regarding the ability to mix GPI-2 and MPI code within the same parallel program.

3 Example: solving a linear system of equations using an Jacobi iterative scheme

In the present document we consider the example of solving a linear system of equations in parallel using an Jacobi iterative scheme. Given a matrix A and a vector b, this method starts with an initial approximation $x^{(0)}$ for the solution x and generates a sequence of vectors $\{x^{(k)}\}_{k=0}^{\infty}$ that converges to x. More explicitly, at each iteration step k, a new approximation solution $x^{(k)}$ is computed from the previous one computed at the step $k-1$:

$$x^{(k)} = D^{-1}(b - (L+U)x^{(k-1)}),$$

where D, L and U are the diagonal, the strictly lower triangular and the strictly upper triangular parts of A, respectively. This can be re-written in a less compact form as following:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left[b_i - \sum_{\substack{j=0 \\ j \neq i}}^{n-1} a_{ij} x_j^{(k-1)} \right], \forall i = 0, n-1.$$

The procedure is repeated until the approximation error falls down below a given threshold.

Although one could choose a more sophisticated example, we believe that this one offers the advantage of being familiar to a large number of application programmers. Its simplicity allows us to illustrate the sensible points related to interoperability.

3.1 MPI implementation of the Jacobi iterative scheme

We start with an MPI implementation of the Jacobi iterative scheme. Although different parallelization strategies may be applied, we opted here for a simple and easy to understand variant. We use this example as a tool for illustrating interoperability aspects between MPI and GPI-2.

The considered MPI implementation is the one illustrated in the listing 5. The program uses the following data structures:

- n is the matrix dimension,
- a is a $n \times n$ square matrix of doubles,
- b is an array of doubles of size n,
- x is the current solution approximation,
- x_{new} is the new solution approximation

The dimension is specified interactively by the user. The input matrix a and the vector b are initialized with some arbitrary values by the process with the rank 0, so that a is symmetric and diagonally dominant, which is a sufficient condition for guaranteeing the convergence of the scheme. The program realizes in the beginning a row-wise distribution of the matrix a and of the vector b across the nodes. It starts with an initial solution approximation x, which is set by the process with the rank 0 and then replicated on all nodes. At any iteration step, each process concurrently computes a sub-vector of the solution approximation x_{new} and after computation, each process sends its local part of the new approximation to the others, using MPI non-blocking operations. The program terminates when a predefined maximum number of iterations is reached or the approximation error is below some given tolerance.

4 Combining GPI-2 and MPI in parallel programs

In this section we illustrate how the initial MPI program given in the listing 5 can be modified such that it mixes GPI-2 with MPI code, by replacing MPI related parts with GPI-2 parts, highlighting the aspects that one should take into consideration when writing mixed-mode GPI-2 programs.

4.1 Installation of GPI-2 with MPI mixed-mode support

Writing parallel programs that are mixing MPI and GPI-2 sections is currently possible due to the ability of GPI-2 to capture the environment of an existing MPI

running instance. For this purpose, the GPI-2 installation script should be given the location of the current MPI installation used by the user. This is possible by using the option –**with-mpi**, as explained in the README file from the GPI-2 source tree from the repository indicated in the section 1, as below:

```
--with-mpi <path_to_mpi_installation>
```

4.2 Environment initialization

When running in mixed-mode, GPI-2 is able to detect at runtime the MPI environment and to setup its own environment based on this. Thus, in mixed-mode GPI-2 is able to deliver similar consistent related information to the users. Particularly, GPI-2 can deliver the same information about the ranks and the number of processes as MPI. In order to be able to do this, MPI must be initialized before GPI-2, as shown in the listing below.

Listing 1: In mixed-mode GPI-2 needs to be initialzed after MPI

```
#include <assert.h>
#include <GASPI.h>
#include <mpi.h>

int main (int argc, char *argv[])
{
  // initialize MPI and GASPI
  MPI_Init (&argc, &argv);
  SUCCESS_OR_DIE (gaspi_proc_init, GASPI_BLOCK);

  // Do work...

  // shutdown GASPI and MPI
  SUCCESS_OR_DIE (gaspi_proc_term, GASPI_BLOCK);
  MPI_Finalize();

  return 0;
}
```

In the above code snippet, **SUCCESS_OR_DIE** is just a convenience macro that prints an error message and exits the program in case the function given as first argument fails when applied to the rest of the arguments (i.e. doesn't return **GASPI_SUCCESS**), as below:

```
#define SUCCESS_OR_DIE(f, args...)                \
```

```
  do                                                    \
  {                                                     \
    gaspi_return_t const r = f (args);                  \
                                                        \
    if (r != GASPI_SUCCESS)                             \
    {                                                   \
      ERROR (gaspi_error_str (r));                      \
    }                                                   \
} while (0)
```

The code snippet from the listing 1 also works when `MPI_Init_thread` is used instead of `MPI_Init` to initialize MPI with support for threads.

It is good practice to always check after initialization if the MPI ranks and the GASPI ranks, as well as the number of processes, are the same in both environments, as done in the listing 2.

Listing 2: GPI-2 uses the same ranks and number of processes as MPI in mixed-mode

```
...
  int my_mpi_rank, n_mpi_procs;
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_mpi_rank);
  MPI_Comm_size (MPI_COMM_WORLD, &n_mpi_procs);

  gaspi_rank_t my_gaspi_rank, n_gaspi_procs;
  SUCCESS_OR_DIE (gaspi_proc_init, GASPI_BLOCK);
  SUCCESS_OR_DIE (gaspi_proc_rank, &my_gaspi_rank);
  SUCCESS_OR_DIE (gaspi_proc_num, &n_gaspi_procs);

  assert(my_mpi_rank == my_gaspi_rank);
  assert(n_mpi_procs == n_gaspi_procs);
...
```

4.3 Starting a parallel application in mixed-mode

Launching a parallel application mixing MPI and GPI-2 code should be done in the same way as when running an MPI standalone application, that is, by invoking the command **mpirun** or similar with the usual parameters. In this case **gaspi_run** should not be used.

4.4 Replacing MPI non-blocking communication with GPI-2 one-sided communication

Porting parts of or an entire MPI application to GPI-2 can be done stepwise, by identifying independent MPI communication blocks and patterns and then, by replacing them with GPI-2 communication blocks. An important rule to follow here is to preserve application's logic. Another aspect to take into account is not to overlap MPI communication with GPI-2 communication sections.

Examining the starting MPI example from listing 5 in appendix, one can remark that a new iteration is started only after a process has received all the missing parts of the current solution approximation and if is ensured that its part of approximation was sent to all peers before overwriting it. With GPI-2, one can take advantage of using one-sided communication and weak synchronization, as will be explained in the following. This means that one could immediately start a new iteration, as soon as the local parts of the approximation vector x are received, without waiting for the send operations to complete. We want to replace the MPI communication at the end of an iteration with GPI one-sided communication. The rest of the code related to initialization and initial data distribution is left unchanged.

Re-writing the program in listing 5 in mixed-mode can be done incrementally. The first step is to initialize the GPI environment, after MPI, like shown in listing 2. The next step is to create two GASPI segments that will be used for communicating the local parts of the solution approximation between processes. The reason for using two segments is to be able to implement a weak-synchronization mechanism, in order to avoid overwriting data or notifications. It may happen that one rank is faster than one of its peers and, while this one is waiting to get all the expected notifications, the first one overwrites its data. In order to prevent such situations, we switch the segments between the iterations. This way, no explicit synchronization is required. The writer process cannot go farther than one iteration, because it must wait for the notifications triggered by the write operations of its peers. This implicit synchronization scheme coupled with the double buffering technique is what we refer to as weak synchronization. This is a common pattern to be used in similar GASPI iterative codes.

The segments are created with **gaspi_segment_create**. Please note that this is an operation that is semantically equivalent to a collective aggregation of **gaspi_segment_alloc**, **gaspi_segment_register** and **gaspi_barrier** involving all of the members of a given group.

Listing 3: Creation of two segments of size $n \times sizeof(double)$

```
gaspi_segment_id_t segment_id_from = 0;
gaspi_segment_id_t segment_id_to = 1;
```

```
SUCCESS_OR_DIE
  ( gaspi_segment_create
  , segment_id_from
  , n * sizeof (double)
  , GASPI_GROUP_ALL
  , GASPI_BLOCK
  , GASPI_MEM_UNINITIALIZED
  );

SUCCESS_OR_DIE
  ( gaspi_segment_create
  , segment_id_to
  , n * sizeof (double)
  , GASPI_GROUP_ALL
  , GASPI_BLOCK
  , GASPI_MEM_UNINITIALIZED
  );
```

The input matrix a and the input vector b are initialized by the process with the rank 0 and distributed exactly as in the original MPI program. The initial solution approximation is set by the process with the rank 0 and broadcasted to all before starting the iterations, again as in the original MPI program. As the loop for calculating successive approximations uses now GPI-2 communication, each process must copy the initial approximation into a segment, before starting the loop:

```
memmove (gaspi_ptr_from, x, n * sizeof (double));
```

The local parts of the new solution approximation x_{new} are computed in the same way and this part of the code is left unchanged, just as in the original MPI program.

The next step is to effectively replace the non-blocking operations. The GASPI standard provides a primitive that realizes the data movement to the remote site, posting at the end of the operation a notification for the remote rank. This primitive is called **gaspi_write_notify** and has the following signature:

```
gaspi_return_t
gaspi_write_notify
  ( const gaspi_segment_id_t segment_id_local
  , const gaspi_offset_t offset_local
  , const gaspi_rank_t rank
  , const gaspi_segment_id_t segment_id_remote
  , const gaspi_offset_t offset_remote
```

```
      , const gaspi_size_t size
      , const gaspi_notification_id_t notification_id
      , const gaspi_notification_t notification_value
      , const gaspi_queue_id_t queue
      , const gaspi_timeout_t timeout_ms
      );
```

Where the parameters are the identifier of the local segment where the data is currently stored, the data offset within this segment (*offset_local*), the target's rank, the remote segment identifier, the offset where to store the data within the remote segment (*offset_remote*), the size of the data to transfer, the notification identifier, a value associated with the notification, the queue identifier where the notification was posted and a timeout value. Thus, the MPI_Isend operations are replaced with gaspi_write_notify operations and executed concurrently as below:

```
// remotely write the local part of the approximation solution
#pragma omp parallel for
for (int dest = 0; dest < n_gaspi_procs ; dest++)
{
  if (dest == my_gaspi_rank)
   continue;
  SUCCESS_OR_DIE
   ( gaspi_write_notify
   , segment_id_to
   , offset * sizeof (double)
   , dest
   , segment_id_to
   , offset * sizeof (double)
   , n_local_rows * sizeof (double)
   , (gaspi_notification_id_t) (my_gaspi_rank)
   , (gaspi_notification_t) (my_gaspi_rank + 1)
   , queue
   , GASPI_BLOCK
   );
}
```

As here we are using GPI-2 one-sided communication, the remote target process is not forced to post a call similar to MPI_Irecv, matching an MPI_Isend operation, in order to announce that it is ready to receive some data. The remote process is not aware that somebody eventually writes something into its memory, until it receives a notification that this has already happened.

The next step is to replace the MPI_Wait routines. In GPI-2, the remote rank should check for locally posted notifications in order to find out relevant

information about the modifications occurred with respect to a segment. Contrary to the MPI variant, which has to wait for an MPI_Isend operation to complete on the sender side, when using GPI-2 communication, only the target side should wait for the completion of a **gaspi_write_notify operation**. For this purpose, the GASPI standard provides the **gaspi_notify_waitsome** and **gaspi_notify_reset** primitives, which are waiting for posted notifications and are resetting arrived notifications, respectively. Please note that these are thread safe routines and multiple threads can be used to wait for notifications, only one being able to atomically reset the value of a notification. The first primitive has the following signature:

```
gaspi_return_t
gaspi_notify_waitsome
( const gaspi_segment_id_t segment_id_local
, const gaspi_notification_id_t notification_begin
, const gaspi_number_t num
, gaspi_notification_id_t* first_id
, const gaspi_timeout_t timeout_ms
);
```

Where the parameters are: the local segment id, the identifier of the first notification to expect, the number of consecutive notifications, the notification identifier that just has arrived and a timeout value for the operation. The application should subsequently reset that notification and retrieve its value with **gaspi_notify_reset**:

```
gaspi_return_t
gaspi_notify_reset
( const gaspi_segment_id_t segment_id_local
, const gaspi_notification_id_t notification_id
, gaspi_notification_t* old_notification_val
);
```

This is a thread-safe atomic operation, guaranteeing that only one thread is able to reset the notification value of the notification passed as the second parameter and related to the segment passed as the first parameter. The old value is returned in the third parameter, *old_notification_val*.

The MPI_Wait operations are replaced with gaspi_notify_waitsome and gaspi_notify_reset operations, as below:

```
// receive notifications upon write operations completion
int completed = 0;
#pragma omp parallel shared (completed)
while (completed < n_gaspi_procs - 1)
{
```

```
    gaspi_notification_id_t received_notification;
    gaspi_return_t rv = gaspi_notify_waitsome
      ( segment_id_to
      , 0
      , n_gaspi_procs
      , &received_notification
      , GASPI_TEST
      );

    // use test so that all threads come out of while
    if (rv == GASPI_TIMEOUT)
      continue;

    gaspi_notification_t value;
    SUCCESS_OR_DIE
      ( gaspi_notify_reset
      , segment_id_to
      , received_notification
      , &value
      );
    if (value)
    {
       #pragma omp atomic
       completed++;
    }
  }
```

At the beginning of a new iteration, the segment identifiers are swapped, in order to use double buffering. The variables containing the current and the previous approximation solutions, x and x_{new}, pointing to addresses allocated within these segments, are also swapped.

For compiling the program, the user should make sure to link against the GPI-2 library, eventually adapting the paths. The program is launched using the mpirun command (or similar), just as in the case of the original MPI program.

4.5 Using application provided memory for segments in applications mixing GPI-2 and MPI

Although the mixed-mode solution described in the previous subsection potentially improves the original MPI code, by using one-sided communication, weak synchronization and multi-threaded notification waiting, it can still be improved by using the new features offered by GPI-2 (starting with the version 1.3). Particularly

interesting is the possibility to allow a user to provide already allocated memory for the segments. In the above described code, each process allocates some local memory for storing a copy of the approximation solution x. Initially, the start approximation solution is communicated to the peers by the process with rank 0, using a broadcast operation. After each process creates the two segments that are used for GPI-2 communication, the content of x is copied into the corresponding segment part. One can avoid this by telling GPI-2 to reuse the buffer used in the MPI broadcast, after the operation finished, as memory allocated for a segment. This is possible due to the introduction into the GASPI standard of the primitives **gaspi_segment_bind** and **gaspi_segment_use**, following a recommendation of the GASPI Forum [2]. The first operation is a synchronous local blocking procedure that binds a segment to user provided memory. The second primitive is semantically equivalent to a collective aggregation of **gaspi_segment_bind**, **gaspi_segment_register** and **gaspi_gaspi_barrier**, involving all members of a given group. If the communication infrastructure was not established for all group members beforehand, this operation accomplishes this as well.

We can modify the mixed variant of the Jacobi scheme so that one can take advantage of the aforementioned feature. Each process will allocate memory for the approximate solutions x and x_{new} and GPI is instructed to bind these pieces of memory to the segments. This is achieved by using the mentioned operation **gaspi_segment_use**, as in the example below:

Listing 4: Use gaspi_segment_use instead of gaspi_segment_create

```
gaspi_segment_id_t segment_id_from = 0;
gaspi_segment_id_t segment_id_to = 1;

SUCCESS_OR_DIE
  ( gaspi_segment_use
  , segment_id_from
  , x
  , n*sizeof (double)
  , GASPI_GROUP_ALL
  , GASPI_BLOCK
  , 0
  );

double* x_new = new double[n];
SUCCESS_OR_DIE
  ( gaspi_segment_use
  , segment_id_to
  , x_new
  , n*sizeof (double)
```

```
, GASPI_GROUP_ALL
, GASPI_BLOCK
, 0
);
```

The clear advantage compared to the mixed-mode variant presented in the subsection 4.4 is the direct access to the segment allocated memory (avoiding thus calling **gaspi_segment_ptr**). The other advantage is the reuse in GPI-2 communication of the buffer x, which was previously used in an MPI broadcast operation.

One important aspect to point here is that GPI-2 will not automatically free the user allocated memory for the segments in this case, therefore it is the user's responsibility to free it after termination.

4.6 Using GPI-2 segment allocated memory as an MPI communication buffer

An alternative to the previous solution that is using a buffer already used in MPI communication as memory for a GPI-2 segment is the reverse situation: use the memory allocated for a GASPI segment as a buffer for MPI communication, avoiding thus explicitly copying the content of an MPI buffer into a segment. The modifications with respect to the implementation evoked in the subsection 4.4 are straightforward. After initializing the MPI and GPI-2 environments and after initializing the matrix a and the vector b in the same way as done in the referred listing, firstly the two segments necessary for GPI-2 communication are created and instead of allocating new memory for the initial approximation x, just let this point to some address within the first segment:

```
x = (double*)gaspi_ptr_from;
```

Then, the process with the rank 0 initializes the start approximation solution directly in the corresponding segment allocated memory, eliminating the necessity of the memove operation. A subsequent MPI broadcast realizes the transfer of a copy of the initial approximation directly into the first segment, for each rank, as below:

```
MPI_Bcast (gaspi_ptr_from, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

4.7 Jacobi iterative scheme in mixed-mode calling GPI code from a library

Another possibility for writing interoperable programs is to call GPI-2 code from a library. In the first two examples above, which are mixing GPI-2 code and MPI code, one can entirely encapsulate the GPI-2 code into a library function and call it in an MPI program. One may for example define and implement a function `jacobi`, taking as parameters the matrix dimension, the local parts of the matrix a and of the vector b, the local x and x_{new} vectors, the maximum number of iterations and the tolerance constant. Each rank calls this library function and is assumed that before calling it, the main program has already initialized initialized the MPI and GPI-2 environments and distributed the matrix a, the vector b and the initial solution approximation.

With these, the main program is as described in the subsections 4.4 or 4.5, with the difference that apart the functions related to GPI-2 initialization and termination, the rest of the GPI-2 code was encapsulated into a library function that is implementing the iterative scheme for a rank.

5 Conclusions

The GASPI API has been designed to coexist with MPI, aiming at providing interoperability with MPI in order to allow for incremental porting of existing applications. In the current document we present aspects related to the interoperability between GPI-2 and MPI. Here, we try via concrete examples to show different modalities for mixing MPI and GPI-2 within the same program, starting from an existing MPI program, incrementally modifying it by replacing MPI routines with GPI-2 routines and trying in the same time to use common patterns and features specific to GPI-2 programming. The described examples show that porting an MPI program to GPI-2 can be done incrementally and smoothly.

1 Appendix

Listing 5: MPI implementation of an Jacobi iterative scheme

```
#include "mpi.h"
#include "init_data.h"
#include <assert.h>
#include "dist.h"
#include "omp.h"

int main(int argc, char* argv[])
{
  double *a, *b;
  int n, my_mpi_rank, n_mpi_procs;

  int mpisupport;
  MPI_Init_thread (&argc, &argv, MPI_THREAD_MULTIPLE, &mpisupport);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_mpi_rank);
  MPI_Comm_size (MPI_COMM_WORLD, &n_mpi_procs);

  if (my_mpi_rank == 0)
    n = init_input_data (&a, &b);

  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

  double* x = (double*) calloc (n, sizeof (double));
  if (my_mpi_rank == 0)
    init_solution (x, n);

  int* sendcounts = (int*) calloc (n_mpi_procs, sizeof (int));
  int* displs = (int*) calloc (n_mpi_procs, sizeof (int));

  comp_mult_counts_and_displs
    (n, n, n_mpi_procs, &sendcounts, &displs);

  int n_local_rows = get_num_rows (my_mpi_rank, n, n_mpi_procs);
  double* local_a =
    (double*) calloc (n_local_rows * n, sizeof (double));

  MPI_Scatterv ( a, sendcounts, displs, MPI_DOUBLE, local_a
               , n_local_rows*n, MPI_DOUBLE, 0, MPI_COMM_WORLD
               );
```

```
comp_counts_and_displs
  (n, n_mpi_procs, &sendcounts, &displs);

double* local_b =
  (double*) calloc (n_local_rows, sizeof (double));
MPI_Scatterv ( b, sendcounts, displs, MPI_DOUBLE, local_b
             , n_local_rows, MPI_DOUBLE, 0, MPI_COMM_WORLD
             );

MPI_Bcast (x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

comp_counts_and_displs
  (n, n_mpi_procs, &sendcounts, &displs);

int offset = get_offset (my_mpi_rank, n, n_mpi_procs);
double* x_new = (double*)calloc (n, sizeof (double));

MPI_Request* send_requests =
  (MPI_Request*) calloc (n_mpi_procs-1, sizeof (MPI_Request));
MPI_Request* recv_requests =
  (MPI_Request*) calloc (n_mpi_procs-1, sizeof (MPI_Request));
MPI_Status* statuses =
  (MPI_Status*) calloc (n_mpi_procs-1, sizeof (MPI_Status));

int n_iterations = 0;
do
{
  if (n_iterations > 0)
    std::swap (x, x_new);

  #pragma omp parallel for
  for (int i = 0; i < n_local_rows; i++)
  {
    double d = local_b[i];
    for (int j = 0; j<i + offset; j++)
      d -= local_a[i*n+j] * x[j];
    for (int j = i + offset + 1; j < n; j++)
      d -= local_a[i*n+j] * x[j];
    x_new[i + offset] = d/local_a[i*n + i + offset];
  }

  #pragma omp parallel for
  for (int rank = 0; rank < n_mpi_procs; rank++)
```

```
  {
    if (rank == my_mpi_rank)
      continue;
    MPI_Isend ( x_new + get_offset (my_mpi_rank, n, n_mpi_procs)
              , get_num_rows (my_mpi_rank, n, n_mpi_procs)
              , MPI_DOUBLE
              , rank, 100, MPI_COMM_WORLD
              , &send_requests[rank<my_mpi_rank?rank:rank-1]
              );
  }

  #pragma omp parallel for
  for (int rank = 0; rank < n_mpi_procs; rank++)
  {
    if (rank == my_mpi_rank)
      continue;
    MPI_Irecv( x_new + get_offset (rank, n, n_mpi_procs)
              , get_num_rows (rank, n, n_mpi_procs), MPI_DOUBLE
              , rank, 100, MPI_COMM_WORLD
              , &recv_requests[rank<my_mpi_rank?rank:rank-1]
              );

  }
  MPI_Waitall (n_mpi_procs-1, recv_requests, statuses);
  MPI_Waitall (n_mpi_procs-1, send_requests, statuses);

} while (n_iterations++ < MAX_ITER && error (x, x_new, n) >= TOL);


if (my_mpi_rank == 0)
{
  free (a);
  free (b);
}

free (sendcounts);
free (displs);
free (send_requests);
free (recv_requests);
free (local_a);
free (local_b);
free (x);
free (x_new);
```

```
  MPI_Finalize();

  return 0;
}
```

Bibliography

[1] Rui Machado, Tiberiu Rotaru, Mirko Rahn, and Valeria Bartsch. Guide to porting mpi applications to gpi-2, 2016.

[2] Rui Machado, Mirko Rahn, Daniel Gruenewald, and Valeria Bartsch. Gaspi proposal: Memory provided by applications.