# GCL

Generated by Doxygen 1.7.1

Mon Nov 14 2011 17:21:16

# Contents

# Chapter 1

# Generic Communicatin Library Reference Manual

# Chapter 2

# Introduction

The Generic Communication Library (GCL) is organized in three levels that will be described below here. The general idea is, however, to produce a **library of communication patterns** each of which providing the three layers. Each pattern is associated with a *topology class* which is used to put processes/threads in relation with each other. It is not necessary that different patterns share common concepts and interfaces, even though uniformity should be a target. Given a pattern, up to three versions of it may exist, each corresponding to a level of the abstraction hierarchy. They can be used independently of each other.

The main idea is that level 3 is the lowest level which maps to current communication mechanisms, like MPI. It can be roughly identified with the *transport layer* in a communication network. It assumes data is transferred though a **high-latency medium** so **buffering is essential**. Actually what level 3 does is to **take already filled buffers and exchange them** according to the communication pattern.

Level 3 can be used in other scenarios, but it can result in poor performance since unnecessary copies may be required, since the **user must buffer data explicitly**.

**Level 2 assumes certain data layouts and distributions**, and perform the patterns according to this knowledge. As an example, consider the halo exchange in a stencil application for regular grids. The big problem grids are partitioned into smaller grids, each of those processed by a process, a thread, or something else. This is the domain decomposition scenario. The decomposition is assumed to be uniform, so each process/thread has the same abstraction over the data. Level 2 knows about it and **takes the description of the data to be exchanged and performs the data exchange**. It may or no use the level 3, since buffering may be not required. Several communication mechanisms allow for semi-shared memory view of the address space. This may require the grids **memory be registered with communication pattern**, so that a mapping of the addresses may allow remote writes of elements. Since this is generic, the registration should happen anyway, since the api should work on every *transport layer*. This is indication of the problem: **since communication mechanisms and tools are so diverse, the generic API can end up in requiring lot of information to deal with all possibilities.** Alternatively, **the choice may be to provide initialization procedures that are architecture dependent**. The initialization problem is probably the more complex to approach in a generic way, and I think we need to be very pragmatic.

**Level 1** is similar to level2, only that the **user must provide**, instead of description of memory areas, **functions to extract and insert data in those areas**. Actually, level 2 can be implemented with level 1 (Now, the story of the levels comes from early discussions, and should change into something more reasonable). In figure a showed an container+iterator interface, which seems to be the most promising. The iterators must contain all the logic to pick the right values, thus jumping around memory as needed. This approach may be used to produce communication patters for more general applications than stencil application on regular grids. **Level 1, as level 2, can take advantage of level 3 when used on top of a high latency interconnect**.

# Chapter 3

# GCL Level 3

# Chapter 4

# GCL Level 2

# Chapter 5

# GCL Level 1

# Chapter 6

# Module Index

## 6.1 Modules

Here is a list of all modules:

# Chapter 7

# Namespace Index

## 7.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

# Chapter 8

# Class Index

## 8.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

# Chapter 9

# File Index

## 9.1 File List

Here is a list of all documented files with brief descriptions:

# Chapter 10

# Module Documentation

## 10.1 Processor Grid Concept

A process grid is a class that provides the interfaces of the generic class (concdept) proc_grid_2D_concept for 2D case and proc_grid_3D_concept for 3D case

# Chapter 11

# Namespace Documentation

## 11.1 GCL Namespace Reference

### Classes

- class Halo_Exchange_2D
- class _2D_process_grid_t
- class MPI_2D_process_grid_t

### 11.1.1 Detailed Description

### 11.1.2 Concepts in GCL

All library classes, functions, and objects will reside in this namespace.

# Chapter 12

# Class Documentation

## 12.1 GCL::_2D_process_grid_t< CYCLIC > Class Template Reference

```
#include <proc_grids_2D.h>
```

### Public Member Functions

- _2D_process_grid_t (int P, int pid)
- void create (int P, int pid)
- void dims (int &t_R, int &t_C) const
- void coords (int &t_R, int &t_C) const
- template<int I, int J>
  int proc () const
- int proc (int I, int J) const

### 12.1.1 Detailed Description

**template<bool CYCLIC> class GCL::_2D_process_grid_t< CYCLIC >**

Class that provides a generic 2D process grid from a linear process distribution. Given a contiguos range of P processes, from 0 to P-1, this class provide a distribution of these processes as a 2D grid by row in the best possible aspect ratio. For example, processes 0,1,2,3,4,5 will be layed out as

```
0 1
2 3
4 5
```

**Template Parameters**

***CYCLIC*** is a boolean template parameter that, if true, specifies the grid is clyclic (in both dimensions)
This is a process grid matching the Processor Grid Concept concept

**Examples:**

Halo_Exchange_test.cpp, and Halo_Exchange_test_2.cpp.

### 12.1.2 Constructor & Destructor Documentation

#### 12.1.2.1 template<bool CYCLIC> GCL::_2D_process_grid_t< CYCLIC >::_2D_process_grid_t ( int *P*, int *pid* ) `[inline]`

Constructor that takes the number of processes and the caller ID to produce the grid

**Parameters**

`[in]` ***P*** Number of processes that will make the grid

`[in]` ***pid*** Number of processes that will make the grid

### 12.1.3 Member Function Documentation

#### 12.1.3.1 template<bool CYCLIC> void GCL::_2D_process_grid_t< CYCLIC >::coords ( int & *t_R*, int & *t_C* ) const `[inline]`

Returns in t_R and t_C the coordinates ot the caller process in the grid AS PRESCRIBED BY THE CONCEPT

**Parameters**

`[out]` ***t_R*** Coordinate in first dimension

`[out]` ***t_C*** Coordinate in second dimension

#### 12.1.3.2 template<bool CYCLIC> void GCL::_2D_process_grid_t< CYCLIC >::create ( int *P*, int *pid* ) `[inline]`

Function to create the grid. This can be called in case the grid is default constructed. Its direct use is discouraged

**Parameters**

`[in]` ***P*** Number of processes that will make the grid

`[in]` ***pid*** Number of processes that will make the grid

#### 12.1.3.3 template<bool CYCLIC> void GCL::_2D_process_grid_t< CYCLIC >::dims ( int & *t_R*, int & *t_C* ) const `[inline]`

Returns in t_R and t_C the lenght of the dimensions of the process grid AS PRESCRIBED BY THE CONCEPT

**Parameters**

`[out]` ***t_R*** Number of elements in first dimension

`[out]` ***t_C*** Number of elements in second dimension

#### 12.1.3.4 template<bool CYCLIC> int GCL::_2D_process_grid_t< CYCLIC >::proc ( int *I*, int *J* ) const `[inline]`

Returns the process ID of the process with relative coordinates (I,J) with respect to the caller process AS PRESCRIBED BY THE CONCEPT

**Parameters**

`[in]` ***I*** Relative coordinate in the first dimension

`[in]` ***J*** Relative coordinate in the seocnd dimension

**Returns**

The process ID of the required process

#### 12.1.3.5 template<bool CYCLIC> template<int I, int J> int GCL::_2D_process_grid_t< CYCLIC >::proc ( ) const `[inline]`

Returns the process ID of the process with relative coordinates (I,J) with respect to the caller process AS PRESCRIBED BY THE CONCEPT

**Template Parameters**

***I*** Relative coordinate in the first dimension

***J*** Relative coordinate in the seocnd dimension

**Returns**

The process ID of the required process

The documentation for this class was generated from the following file:

- L3/include/proc_grids_2D.h

## 12.2 GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE > Class Template Reference

`#include <Halo_Exchange_2D.h>`

### Public Member Functions

- Halo_Exchange_2D (PROC_GRID _pg)
- PROC_GRID const & proc_grid ()
- void register_send_to_buffer (void ∗p, int s, int I, int J)
- template<int I, int J>
  void register_send_to_buffer (void ∗p, int s)
- void register_receive_from_buffer (void ∗p, int s, int I, int J)
- template<int I, int J>
  void register_receive_from_buffer (void ∗p, int s)
- void set_send_to_size (int s, int I, int J)

- template<int I, int J>
  void set_send_to_size (int s) const
- void set_receive_from_size (int s, int I, int J) const
- template<int I, int J>
  void set_receive_from_size (int s) const
- void exchange ()

### 12.2.1 Detailed Description

**template<typename PROC_GRID, int ALIGN_SIZE = 1> class GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE >**

Class to instantiate, define and run a regular cyclic and acyclic halo exchange pattern in 2D. By regular it is intended that the amount of data sent and received during the execution of the pattern is known by all participants to the comunciation without communication. More specifically, the ampunt of data received is decided before the execution of the pattern. If a different ampunt of data is received from some process the behavior is undefined.

Given a process (i,j), we can define $s_{ij}^{mn}$ and $r_{ij}^{mn}$ as the data sent and received from process (i,j) to/from process (i+m, j+m), respectively. For this pattern m and n are supposed to be in the range -1, 0, +1.

When executing the Halo_Exchange_2D pattern, the requirement is that

$$r_{ij}^{mn} = s_{i+m,j+n}^{-m,-n}$$

.

**Template Parameters**

***PROG_GRID*** Processor Grid type. An object of this type will be passed to constructor.

***ALIGN*** integer parameter that specify the alignment of the data to used. UNUSED IN CURRENT VERSION
Pattern for regular cyclic and acyclic halo exchange pattern in 2D The communicating processes are arganized in a 2D grid. Given a process, neighbors processes are located using relative coordinates. In the next diagram, the given process is (0,0) while the neighbors are indicated with their relative coordinates.

```
-------------------------
|        |        |        |
| -1,-1  | -1,0   | -1,1   |
|        |        |        |
-------------------------
|        |        |        |
|  0,-1  |  0,0   |  0,1   |
|        |        |        |
-------------------------
|        |        |        |
|  1,-1  |  1,0   |  1,1   |
|        |        |        |
-------------------------
```

The pattern is cyclic or not bepending on the process grid passed to it. The cyclicity may be on only one dimension. An example of use of the pattern is given below

```
int iminus;
int iplus;
int jminus;
int jplus;
int iminusjminus;
int iplusjminus;
int iminusjplus;
```

```
        int iplusjplus;

        int iminus_r;
        int iplus_r;
        int jminus_r;
        int jplus_r;
        int iminusjminus_r;
        int iplusjminus_r;
        int iminusjplus_r;
        int iplusjplus_r;

        typedef GCL::_2D_proc_grid_t grid_type;

        grid_type pg(P,my_id);

        GCL::Halo_Exchange_2D<grid_type> he(pg);

        he.register_send_to_buffer<-1,-1>(&iminusjminus, sizeof(int));
        he.register_send_to_buffer<-1, 1>(&iminusjplus, sizeof(int));
        he.register_send_to_buffer< 1,-1>(&iplusjminus, sizeof(int));
        he.register_send_to_buffer< 1, 1>(&iplusjplus, sizeof(int));
        he.register_send_to_buffer<-1, 0>(&iminus, sizeof(int));
        he.register_send_to_buffer< 1, 0>(&iplus, sizeof(int));
        he.register_send_to_buffer< 0,-1>(&jminus, sizeof(int));
        he.register_send_to_buffer< 0, 1>(&jplus, sizeof(int));

        he.register_receive_from_buffer<-1,-1>(&iminusjminus_r, sizeof(int));
        he.register_receive_from_buffer<-1, 1>(&iminusjplus_r, sizeof(int));
        he.register_receive_from_buffer< 1,-1>(&iplusjminus_r, sizeof(int));
        he.register_receive_from_buffer< 1, 1>(&iplusjplus_r, sizeof(int));
        he.register_receive_from_buffer<-1, 0>(&iminus_r, sizeof(int));
        he.register_receive_from_buffer< 1, 0>(&iplus_r, sizeof(int));
        he.register_receive_from_buffer< 0,-1>(&jminus_r, sizeof(int));
        he.register_receive_from_buffer< 0, 1>(&jplus_r, sizeof(int));

        he.exchange();
```

A running example can be found in the included example.

**Examples:**

Halo_Exchange_test.cpp, and Halo_Exchange_test_2.cpp.

## 12.2.2 Constructor & Destructor Documentation

### 12.2.2.1 template<typename PROC_GRID, int ALIGN_SIZE = 1> GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE >::Halo_Exchange_2D ( PROC_GRID *_pg* ) `[inline, explicit]`

Constructor that takes the process grid. Must be executed by all the processes in the grid. It is not possible to change the process grid once the pattern has beeninstantiated.

## 12.2.3 Member Function Documentation

### 12.2.3.1 template<typename PROC_GRID, int ALIGN_SIZE = 1> void GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE >::exchange ( ) `[inline]`

When called this function executes the communication pattern, that is, send all the send-buffers to the correspondinf receive-buffers. When the function returns the data in receive buffers can be safely accessed.

#### 12.2.3.2 template<typename PROC_GRID, int ALIGN_SIZE = 1> PROC_GRID const& GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE >::proc_grid ( ) `[inline]`

Returns the processor grid (as const reference) been used in construction

#### 12.2.3.3 template<typename PROC_GRID, int ALIGN_SIZE = 1> template<int I, int J> void GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE >::register_receive_from_buffer ( void ∗ *p*, int *s* ) `[inline]`

Function to register buffers for received data with the communication patter. Values I and J are coordinates relative to calling process and the buffer is the container for the data to be received from that process. The amount of data is specified as number of bytes. It is possible to override the previous pointer by re-registering a new pointer with a given source.

**Template Parameters**

- ***I*** Relative coordinates of the receiving process along the first dimension
- ***J*** Relative coordinates of the receiving process along the second dimension

**Parameters**

- `[in]` ***p*** Pointer to the first element of type T where to put received data
- `[in]` ***s*** Number of bytes (not number of elements) expected to be received. This is the data that is assumed to arrive. If less data arrives, the behaviour is undefined.

#### 12.2.3.4 template<typename PROC_GRID, int ALIGN_SIZE = 1> void GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE >::register_receive_from_buffer ( void ∗ *p*, int *s*, int *I*, int *J* ) `[inline]`

Function to register buffers for received data with the communication patter. Values I and J are coordinates relative to calling process and the buffer is the container for the data to be received from that process. The amount of data is specified as number of bytes. It is possible to override the previous pointer by re-registering a new pointer with a given source.

**Parameters**

- `[in]` ***p*** Pointer to the first element of type T where to put received data
- `[in]` ***s*** Number of bytes (not number of elements) expected to be received. This is the data that is assumed to arrive. If less data arrives, the behaviour is undefined.
- `[in]` ***I*** Relative coordinates of the receiving process along the first dimension
- `[in]` ***J*** Relative coordinates of the receiving process along the second dimension

#### 12.2.3.5 template<typename PROC_GRID, int ALIGN_SIZE = 1> void GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE >::register_send_to_buffer ( void ∗ *p*, int *s*, int *I*, int *J* ) `[inline]`

Function to register send buffers with the communication patter. Values I and J are coordinates relative to calling process and the buffer is the container for the data to be sent to that process. The amount of data is specified as number of bytes. It is possible to override the previous pointer by re-registering a new pointer with a given destination.

**Parameters**

`[in]` ***p*** Pointer to the first element of type T to send

`[in]` ***s*** Number of bytes (not number of elements) to be send. In any case this is the amount of data sent.

`[in]` ***I*** Relative coordinates of the receiving process along the first dimension

`[in]` ***J*** Relative coordinates of the receiving process along the second dimension

#### 12.2.3.6 template<typename PROC_GRID, int ALIGN_SIZE = 1> template<int I, int J> void GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE >::register_send_to_buffer ( void ∗ *p*, int *s* ) `[inline]`

Function to register send buffers with the communication patter. Values I and J are coordinates relative to calling process and the buffer is the container for the data to be sent to that process. The amount of data is specified as number of bytes. It is possible to override the previous pointer by re-registering a new pointer with a given destination.

**Template Parameters**

***I*** Relative coordinates of the receiving process along the first dimension

***J*** Relative coordinates of the receiving process along the second dimension

**Parameters**

`[in]` ***p*** Pointer to the first element of type T to send

`[in]` ***s*** Number of bytes (not number of elements) to be send. In any case this is the amount of data sent.

#### 12.2.3.7 template<typename PROC_GRID, int ALIGN_SIZE = 1> template<int I, int J> void GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE >::set_receive_from_size ( int *s* ) const `[inline]`

Function to set receive buffers sizes if the size must be updated from a previous registration. The same pointer passed during registration will be used to receive data. It is possible to override the previous pointer by re-registering a new pointer with a given source. Values I and J are coordinates relative to calling process and the buffer is the container for the data to be sent to that process. The amount of data is specified as number of bytes.

**Template Parameters**

***I*** Relative coordinates of the receiving process along the first dimension

***J*** Relative coordinates of the receiving process along the second dimension

**Parameters**

`[in]` ***s*** Number of bytes (not number of elements) to be packed.

#### 12.2.3.8 template<typename PROC_GRID, int ALIGN_SIZE = 1> void GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE >::set_receive_from_size ( int *s*, int *I*, int *J* ) const `[inline]`

Function to set receive buffers sizes if the size must be updated from a previous registration. The same pointer passed during registration will be used to receive data. It is possible to override the previous pointer by re-registering a new pointer with a given source. Values I and J are coordinates relative to calling process and the buffer is the container for the data to be sent to that process. The amount of data is specified as number of bytes.

**Parameters**

`[in]` ***s*** Number of bytes (not number of elements) to be packed.
`[in]` ***I*** Relative coordinates of the receiving process along the first dimension
`[in]` ***J*** Relative coordinates of the receiving process along the second dimension

#### 12.2.3.9 template<typename PROC_GRID, int ALIGN_SIZE = 1> void GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE >::set_send_to_size ( int *s*, int *I*, int *J* ) `[inline]`

Function to set send buffers sizes if the size must be updated from a previous registration. The same pointer passed during registration will be used to send data. It is possible to override the previous pointer by re-registering a new pointer with a given destination. Values I and J are coordinates relative to calling process and the buffer is the container for the data to be sent to that process. The amount of data is specified as number of bytes.

**Parameters**

`[in]` ***s*** Number of bytes (not number of elements) to be sent.
`[in]` ***I*** Relative coordinates of the receiving process along the first dimension
`[in]` ***J*** Relative coordinates of the receiving process along the second dimension

#### 12.2.3.10 template<typename PROC_GRID, int ALIGN_SIZE = 1> template<int I, int J> void GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE >::set_send_to_size ( int *s* ) const `[inline]`

Function to set send buffers sizes if the size must be updated from a previous registration. The same pointer passed during registration will be used to send data. It is possible to override the previous pointer by re-registering a new pointer with a given destination. Values I and J are coordinates relative to calling process and the buffer is the container for the data to be sent to that process. The amount of data is specified as number of bytes.

**Template Parameters**

***I*** Relative coordinates of the receiving process along the first dimension
***J*** Relative coordinates of the receiving process along the second dimension

**Parameters**

`[in]` ***s*** Number of bytes (not number of elements) to be sent.

The documentation for this class was generated from the following files:

- L3/include/Halo_Exchange.h
- L3/include/Halo_Exchange_2D.h

## 12.3 GCL::MPI_2D_process_grid_t< CYCLIC > Class Template Reference

`#include <proc_grids_2D.h>`

### Public Member Functions

- MPI_2D_process_grid_t (MPI_Comm comm)
- void create (MPI_Comm comm)
- void dims (int &t_R, int &t_C) const
- void coords (int &t_R, int &t_C) const
- template<int I, int J>  
  int proc () const
- int proc (int I, int J) const

### 12.3.1 Detailed Description

**template<bool CYCLIC> class GCL::MPI_2D_process_grid_t< CYCLIC >**

Class that provides a representation of a 2D process grid given an MPI CART It requires the MPI CART to be defined before the grid is created

**Template Parameters**

***CYCLIC*** is a boolean template parameter that, if true, specifies the grid is clyclic (in both dimensions)
This is a process grid matching the Processor Grid Concept concept

**Examples:**

Halo_Exchange_test.cpp, and Halo_Exchange_test_2.cpp.

### 12.3.2 Constructor & Destructor Documentation

#### 12.3.2.1 template<bool CYCLIC> GCL::MPI_2D_process_grid_t< CYCLIC >::MPI_2D_process_grid_t ( MPI_Comm *comm* ) `[inline]`

Constructor that takes an MPI CART communicator, already configured, and use it to set up the process grid.

**Parameters**

***comm*** MPI Communicator describing the MPI 2D computing grid

## 12.3.3 Member Function Documentation

### 12.3.3.1 template<bool CYCLIC> void GCL::MPI_2D_process_grid_t< CYCLIC >::coords ( int & *t_R*, int & *t_C* ) const `[inline]`

Returns in t_R and t_C the coordinates ot the caller process in the grid AS PRESCRIBED BY THE CONCEPT

**Parameters**

`[out]` ***t_R*** Coordinate in first dimension

`[out]` ***t_C*** Coordinate in second dimension

### 12.3.3.2 template<bool CYCLIC> void GCL::MPI_2D_process_grid_t< CYCLIC >::create ( MPI_Comm *comm* ) `[inline]`

Function to create the grid. This can be called in case the grid is default constructed. Its direct use is discouraged

**Parameters**

***comm*** MPI Communicator describing the MPI 2D computing grid

### 12.3.3.3 template<bool CYCLIC> void GCL::MPI_2D_process_grid_t< CYCLIC >::dims ( int & *t_R*, int & *t_C* ) const `[inline]`

Returns in t_R and t_C the lenght of the dimensions of the process grid AS PRESCRIBED BY THE CONCEPT

**Parameters**

`[out]` ***t_R*** Number of elements in first dimension

`[out]` ***t_C*** Number of elements in second dimension

### 12.3.3.4 template<bool CYCLIC> int GCL::MPI_2D_process_grid_t< CYCLIC >::proc ( int *I*, int *J* ) const `[inline]`

Returns the process ID of the process with relative coordinates (I,J) with respect to the caller process AS PRESCRIBED BY THE CONCEPT

**Parameters**

`[in]` ***I*** Relative coordinate in the first dimension

`[in]` ***J*** Relative coordinate in the seocnd dimension

**Returns**

The process ID of the required process

#### 12.3.3.5 template<bool CYCLIC> template<int I, int J> int GCL::MPI_2D_process_grid_t< CYCLIC >::proc ( ) const `[inline]`

Returns the process ID of the process with relative coordinates (I,J) with respect to the caller process AS PRESCRIBED BY THE CONCEPT

**Template Parameters**

***I*** Relative coordinate in the first dimension

***J*** Relative coordinate in the seocnd dimension

**Returns**

The process ID of the required process

The documentation for this class was generated from the following file:

- L3/include/proc_grids_2D.h

## 12.4 proc_grid_2D_concept Struct Reference

```
#include <proc_grid_2D.h>
```

### Public Member Functions

- void dims (int &t_R, int &t_C) const
- void coords (int &t_R, int &t_C) const
- template<int I, int J>
  int proc () const
- int proc (int I, int J) const

### 12.4.1 Detailed Description

class proc_grid_2D_concept This is not a real class but only a template to illustrate the concept of a 2D process grid Given a contiguos range of P processes, from 0 to P-1, this class provide a distribution of these processes as a 2D grid by row. For example, processes 0,1,2,3,4,5 will be layed out as

### 12.4.2 Member Function Documentation

#### 12.4.2.1 void proc_grid_2D_concept::coords ( int & *t_R*, int & *t_C* ) const `[inline]`

Returns in t_R and t_C the coordinates ot the caller process in the grid

**Parameters**

`[out]` ***t_R*** Coordinate in first dimension

`[out]` ***t_C*** Coordinate in second dimension

#### 12.4.2.2 void proc_grid_2D_concept::dims ( int & *t_R*, int & *t_C* ) const `[inline]`

Returns in t_R and t_C the lenght of the dimensions of the process grid

**Parameters**

`[out]` ***t_R*** Number of elements in first dimension
`[out]` ***t_C*** Number of elements in second dimension

#### 12.4.2.3 template<int I, int J> int proc_grid_2D_concept::proc ( ) const `[inline]`

Returns the process ID of the process with relative coordinates (I,J) with respect to the caller process

**Template Parameters**

***I*** Relative coordinate in the first dimension
***J*** Relative coordinate in the seocnd dimension

**Returns**

The process ID of the required process

#### 12.4.2.4 int proc_grid_2D_concept::proc ( int *I*, int *J* ) const `[inline]`

Returns the process ID of the process with relative coordinates (I,J) with respect to the caller process

**Parameters**

`[in]` ***I*** Relative coordinate in the first dimension
`[in]` ***J*** Relative coordinate in the seocnd dimension

**Returns**

The process ID of the required process

The documentation for this struct was generated from the following file:

- docs/concepts/proc_grid_2D.h

# Chapter 13

# File Documentation

## 13.1 L3/include/Halo_Exchange_2D.h File Reference

```
#include <boost/mpl/assert.hpp>
#include <assert.h>
#include <GCL.h>
```

### Classes

- class GCL::Halo_Exchange_2D< PROC_GRID, ALIGN_SIZE >

### Namespaces

- namespace GCL

### 13.1.1 Detailed Description

Pattern for regular cyclic and acyclic halo exchange pattern in 2D The communicating processes are arganized in a 2D grid. Given a process, neighbors processes are located using relative coordinates. In the next diagram, the given process is (0,0) while the neighbors are indicated with their relative coordinates.

```
-------------------------
|        |        |        |
| -1,-1  | -1,0   | -1,1   |
|        |        |        |
-------------------------
|        |        |        |
|  0,-1  |  0,0   |  0,1   |
|        |        |        |
-------------------------
|        |        |        |
|  1,-1  |  1,0   |  1,1   |
|        |        |        |
-------------------------
```

# Chapter 14

# Example Documentation

## 14.1 Halo_Exchange_test.cpp

```
#include <mpi.h>
#include <iostream>
#include <proc_grids_2D.h>
#include <Halo_Exchange_2D.h>
#include <stdio.h>

struct T1 {}; // GCL CYCLIC
struct T2 {}; // GCL not CYCLIC
struct T3 {}; // MPI CYCLIC
struct T4 {}; // MPI not CYCLIC

template <typename T>
struct pgrid;

template <>
struct pgrid<T1> {

  typedef GCL::_2D_process_grid_t<true> grid_type;

  static grid_type instantiate(MPI_Comm comm) {
    int pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    int nprocs;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    return grid_type(nprocs, pid);
  }

};

template <>
struct pgrid<T2> {

  typedef GCL::_2D_process_grid_t<false> grid_type;

  static grid_type instantiate(MPI_Comm comm) {
    int pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    int nprocs;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    return grid_type(nprocs, pid);
  }

};
```

```
template <>
struct pgrid<T3> {

  typedef GCL::MPI_2D_process_grid_t<true> grid_type;

  static grid_type instantiate(MPI_Comm comm) {
    int pid;
    MPI_Comm_rank(GCL::GCL_WORLD, &pid);
    int nprocs;
    MPI_Comm_size(GCL::GCL_WORLD, &nprocs);
    MPI_Comm CartComm;
    int dims[2] = {0,0};
    MPI_Dims_create(nprocs, 2, dims);
    int period[2] = {1, 1};

    std::cout << "@" << GCL::PID << "@ MPI GRID SIZE " << dims[0] << " - " << dim
      s[1] << "\n";

    MPI_Cart_create(GCL::GCL_WORLD, 2, dims, period, false, &CartComm);

    return grid_type(CartComm);
  }

};

template <>
struct pgrid<T4> {

  typedef GCL::MPI_2D_process_grid_t<false> grid_type;

  static grid_type instantiate(MPI_Comm comm) {
    int pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    int nprocs;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm CartComm;
    int dims[2] = {0,0};
    MPI_Dims_create(nprocs, 2, dims);
    int period[2] = {0, 0};

    std::cout << "@" << GCL::PID << "@ MPI GRID SIZE " << dims[0] << " - " << dim
      s[1] << "\n";

    MPI_Cart_create(GCL::GCL_WORLD, 2, dims, period, false, &CartComm);

    return grid_type(CartComm);
  }

};

int main(int argc, char** argv) {

  MPI_Init(&argc, &argv);
  GCL::GCL_Init(argc, argv);

  int iminus;
  int iplus;
  int jminus;
  int jplus;
  int iminusjminus;
  int iplusjminus;
  int iminusjplus;
  int iplusjplus;

  int iminus_r=-1;
  int iplus_r=-1;
  int jminus_r=-1;
  int jplus_r=-1;
  int iminusjminus_r=-1;
  int iplusjminus_r=-1;
  int iminusjplus_r=-1;
  int iplusjplus_r=-1;

  typedef pgrid<T3> test_type;

  test_type::grid_type pg = test_type::instantiate(MPI_COMM_WORLD);

  GCL::Halo_Exchange_2D<test_type::grid_type> he(pg);

  std::cout << "@" << GCL::PID << "@ SEND "
            << &iminus << " - "
            << &iplus << " - "
            << &jminus << " - "
            << &jplus << " - "
            << &iminusjminus << " - "
            << &iplusjminus << " - "
            << &iminusjplus << " - "
            << &iplusjplus << std::endl;
  std::cout << "@" << GCL::PID << "@ RECV "
            << &iminus_r << " - "
            << &iplus_r << " - "
            << &jminus_r << " - "
            << &jplus_r << " - "
            << &iminusjminus_r << " - "
            << &iplusjminus_r << " - "
            << &iminusjplus_r << " - "
            << &iplusjplus_r << std::endl;

  he.register_send_to_buffer<-1,-1>(&iminusjminus, sizeof(int));
  he.register_send_to_buffer<-1, 1>(&iminusjplus, sizeof(int));
  he.register_send_to_buffer< 1,-1>(&iplusjminus, sizeof(int));
  he.register_send_to_buffer< 1, 1>(&iplusjplus, sizeof(int));
  he.register_send_to_buffer<-1, 0>(&iminus, sizeof(int));
  he.register_send_to_buffer< 1, 0>(&iplus, sizeof(int));
  he.register_send_to_buffer< 0,-1>(&jminus, sizeof(int));
  he.register_send_to_buffer< 0, 1>(&jplus, sizeof(int));

  he.register_receive_from_buffer<-1,-1>(&iminusjminus_r, sizeof(int));
  he.register_receive_from_buffer<-1, 1>(&iminusjplus_r, sizeof(int));
  he.register_receive_from_buffer< 1,-1>(&iplusjminus_r, sizeof(int));
  he.register_receive_from_buffer< 1, 1>(&iplusjplus_r, sizeof(int));
  he.register_receive_from_buffer<-1, 0>(&iminus_r, sizeof(int));
  he.register_receive_from_buffer< 1, 0>(&iplus_r, sizeof(int));
  he.register_receive_from_buffer< 0,-1>(&jminus_r, sizeof(int));
  he.register_receive_from_buffer< 0, 1>(&jplus_r, sizeof(int));

  iminus = GCL::PID;
  iplus = GCL::PID;
  jminus = GCL::PID;
  jplus = GCL::PID;
  iminusjminus = GCL::PID;
  iplusjminus = GCL::PID;
  iminusjplus = GCL::PID;
  iplusjplus = GCL::PID;

  he.exchange();

  printf("@%3d@ ----------------\n@%3d@ |%3d |%3d |%3d |\n@%3d@ |%3d |%3d |%3d |\
      n@%3d@ |%3d |%3d |%3d |\n@%3d@ ----------------\n\n",
         GCL::PID,
         GCL::PID, iminusjminus_r, iminus_r, iminusjplus_r,
         GCL::PID, jminus_r, GCL::PID, jplus_r,
         GCL::PID, iplusjminus_r, iplus_r, iplusjplus_r,
         GCL::PID);
  int res = 1;

  res &= (pg.proc(-1,-1) == iminusjminus_r);
  res &= (pg.proc(-1, 0) == iminus_r);
  res &= (pg.proc(-1, 1) == iminusjplus_r);
  res &= (pg.proc( 0,-1) == jminus_r);
  res &= (pg.proc( 0, 1) == jplus_r);
  res &= (pg.proc( 1,-1) == iplusjminus_r);
  res &= (pg.proc( 1, 0) == iplus_r);
  res &= (pg.proc( 1, 1) == iplusjplus_r);

  int final;
  MPI_Reduce(&res, &final, 1, MPI_INT, MPI_LAND, 0, GCL::GCL_WORLD);

  if (GCL::PID==0)
    if (!final) {
      std::cout << "@" << GCL::PID << "@ FAILED!\n";
    } else
      std::cout << "@" << GCL::PID << "@ PASSED!\n";


  MPI_Barrier(GCL::GCL_WORLD);
  MPI_Finalize();
  return !final;
}
```

## 14.2 Halo_Exchange_test_2.cpp

```
#include <mpi.h>
#include <iostream>
#include <proc_grids_2D.h>
#include <Halo_Exchange_2D.h>
#include <stdio.h>
#include <vector>
#include <algorithm>

struct T1 {}; // GCL CYCLIC
struct T2 {}; // GCL not CYCLIC
struct T3 {}; // MPI CYCLIC
struct T4 {}; // MPI not CYCLIC

template <typename T>
struct pgrid;


// THIS TEST DOES NOT WORK WITH CYCLIC GRIDS
template <>
struct pgrid<T1> {

  typedef GCL::_2D_process_grid_t<true> grid_type;

  static grid_type instantiate(MPI_Comm comm) {
    int pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    int nprocs;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    return grid_type(nprocs, pid);
  }

};

template <>
struct pgrid<T2> {
  typedef GCL::_2D_process_grid_t<false> grid_type;

  static grid_type instantiate(MPI_Comm comm) {
    int pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    int nprocs;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    return grid_type(nprocs, pid);
  }

};

// THIS TEST DOES NOT WORK WITH CYCLIC GRIDS
template <>
struct pgrid<T3> {

  typedef GCL::MPI_2D_process_grid_t<true> grid_type;

  static grid_type instantiate(MPI_Comm comm) {
    int pid;
    MPI_Comm_rank(GCL::GCL_WORLD, &pid);
    int nprocs;
    MPI_Comm_size(GCL::GCL_WORLD, &nprocs);
    MPI_Comm CartComm;
    int dims[2] = {0,0};
    MPI_Dims_create(nprocs, 2, dims);
    int period[2] = {1, 1};

    std::cout << "@" << GCL::PID << "@ MPI GRID SIZE " << dims[0] << " - " << dim
      s[1] << "\n";

    MPI_Cart_create(GCL::GCL_WORLD, 2, dims, period, false, &CartComm);

    return grid_type(CartComm);
  }

};

template <>
struct pgrid<T4> {

  typedef GCL::MPI_2D_process_grid_t<false> grid_type;

  static grid_type instantiate(MPI_Comm comm) {
    int pid;
    MPI_Comm_rank(MPI_COMM_WORLD, &pid);
    int nprocs;
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm CartComm;
    int dims[2] = {0,0};
    MPI_Dims_create(nprocs, 2, dims);
    int period[2] = {0, 0};

    std::cout << "@" << GCL::PID << "@ MPI GRID SIZE " << dims[0] << " - " << dim
      s[1] << "\n";

    MPI_Cart_create(GCL::GCL_WORLD, 2, dims, period, false, &CartComm);

    return grid_type(CartComm);
  }

};

int main(int argc, char** argv) {

  MPI_Init(&argc, &argv);
  GCL::GCL_Init(argc, argv);
```

```
int k = 16;
int i,j,N,M;

typedef pgrid<T2> test_type;

test_type::grid_type pg = test_type::instantiate(MPI_COMM_WORLD);

GCL::Halo_Exchange_2D<test_type::grid_type> he(pg);

pg.coords(i,j);
pg.dims(N,M);

std::vector<int> iminus( k+i+j );
std::vector<int> iplus( k+i+j );
std::vector<int> jminus( k+i+j );
std::vector<int> jplus( k+i+j );
std::vector<int> iminusjminus( k+i+j );
std::vector<int> iplusjminus( k+i+j );
std::vector<int> iminusjplus( k+i+j );
std::vector<int> iplusjplus( k+i+j );

std::vector<int> iminus_r( k+i+j-1 );
std::vector<int> iplus_r( k+i+j+1 );
std::vector<int> jminus_r( k+i+j-1 );
std::vector<int> jplus_r( k+i+j+1 );
std::vector<int> iminusjminus_r( k+i+j-2 );
std::vector<int> iplusjminus_r( k+i+j );
std::vector<int> iminusjplus_r( k+i+j );
std::vector<int> iplusjplus_r( k+i+j+2 );

he.register_send_to_buffer<-1,-1>(&iminusjminus[0], (k+i+j)*sizeof(int));
he.register_send_to_buffer<-1, 1>(&iminusjplus[0], (k+i+j)*sizeof(int));
he.register_send_to_buffer< 1,-1>(&iplusjminus[0], (k+i+j)*sizeof(int));
he.register_send_to_buffer< 1, 1>(&iplusjplus[0], (k+i+j)*sizeof(int));
he.register_send_to_buffer<-1, 0>(&iminus[0], (k+i+j)*sizeof(int));
he.register_send_to_buffer< 1, 0>(&iplus[0], (k+i+j)*sizeof(int));
he.register_send_to_buffer< 0,-1>(&jminus[0], (k+i+j)*sizeof(int));
he.register_send_to_buffer< 0, 1>(&jplus[0], (k+i+j)*sizeof(int));

he.register_receive_from_buffer<-1,-1>(&iminusjminus_r[0], (k+i+j-2)*sizeof(int
    ));
he.register_receive_from_buffer<-1, 1>(&iminusjplus_r[0], (k+i+j)*sizeof(int));

he.register_receive_from_buffer< 1,-1>(&iplusjminus_r[0], (k+i+j)*sizeof(int));

he.register_receive_from_buffer< 1, 1>(&iplusjplus_r[0], (k+i+j+2)*sizeof(int))
    ;
he.register_receive_from_buffer<-1, 0>(&iminus_r[0], (k+i+j-1)*sizeof(int));
he.register_receive_from_buffer< 1, 0>(&iplus_r[0], (k+i+j+1)*sizeof(int));
he.register_receive_from_buffer< 0,-1>(&jminus_r[0], (k+i+j-1)*sizeof(int));
he.register_receive_from_buffer< 0, 1>(&jplus_r[0], (k+i+j+1)*sizeof(int));

std::fill(&iminus[0], &iminus[k+i+j], GCL::PID);
std::fill(&iplus[0], &iplus[k+i+j], GCL::PID);
std::fill(&jminus[0], &jminus[k+i+j], GCL::PID);
std::fill(&jplus[0], &jplus[k+i+j], GCL::PID);
std::fill(&iminusjminus[0], &iminusjminus[k+i+j], GCL::PID);
std::fill(&iplusjplus[0], &iplusjplus[k+i+j], GCL::PID);
std::fill(&iplusjminus[0], &iplusjminus[k+i+j], GCL::PID);
std::fill(&iminusjplus[0], &iminusjplus[k+i+j], GCL::PID);

he.exchange();

std::vector<int> res_iminus_r( k+i+j-1 );
std::vector<int> res_iplus_r( k+i+j+1 );
std::vector<int> res_jminus_r( k+i+j-1 );
```

```
std::vector<int> res_jplus_r( k+i+j+1 );
std::vector<int> res_iminusjminus_r( k+i+j-2 );
std::vector<int> res_iplusjminus_r( k+i+j );
std::vector<int> res_iminusjplus_r( k+i+j );
std::vector<int> res_iplusjplus_r( k+i+j+2 );

std::fill(&res_iminus_r[0], &res_iminus_r[k+i+j-1], pg.proc<-1,0>());
std::fill(&res_iplus_r[0], &res_iplus_r[k+i+j+1], pg.proc<1,0>());
std::fill(&res_jminus_r[0], &res_jminus_r[k+i+j-1], pg.proc<0,-1>());
std::fill(&res_jplus_r[0], &res_jplus_r[k+i+j+1], pg.proc<0,1>());
std::fill(&res_iminusjminus_r[0], &res_iminusjminus_r[k+i+j-2], pg.proc<-1,-1>(
    ));
std::fill(&res_iplusjplus_r[0], &res_iplusjplus_r[k+i+j+2], pg.proc<1,1>());
std::fill(&res_iplusjminus_r[0], &res_iplusjminus_r[k+i+j], pg.proc<1,-1>());
std::fill(&res_iminusjplus_r[0], &res_iminusjplus_r[k+i+j], pg.proc<-1,1>());

int res = 1;

if (i>0) {
  res &= std::equal(&iminus_r[0], &iminus_r[k+i+j-1], &res_iminus_r[0]);
  std::cout << GCL::PID << " res = " <<  res << "\n";
}
if (i<N-1) {
  res &= std::equal(&iplus_r[0], &iplus_r[k+i+j+1], &res_iplus_r[0]);
  std::cout << GCL::PID << " res = " <<  res << "\n";
}
if (j>0) {
  res &= std::equal(&jminus_r[0], &jminus_r[k+i+j-1], &res_jminus_r[0]);
  std::cout << GCL::PID << " res = " <<  res << "\n";
}
if (j<M-1) {
  res &= std::equal(&jplus_r[0], &jplus_r[k+i+j+1], &res_jplus_r[0]);
  std::cout << GCL::PID << " res = " <<  res << "\n";
}
if (i>0 && j>0) {
  res &= std::equal(&iminusjminus_r[0], &iminusjminus_r[k+i+j-2], &res_iminusjm
    inus_r[0]);
  std::cout << GCL::PID << " res = " <<  res << "\n";
}
if (i<N-1 && j>0) {
  res &= std::equal(&iplusjminus_r[0], &iplusjminus_r[k+i+j], &res_iplusjminus_
    r[0]);
  std::cout << GCL::PID << " res = " <<  res << "\n";
}
if (i>0 && j<M-1) {
  res &= std::equal(&iminusjplus_r[0], &iminusjplus_r[k+i+j], &res_iminusjplus_
    r[0]);
  std::cout << GCL::PID << " res = " <<  res << "\n";
}
if (i<N-1 && j<M-1) {
  res &= std::equal(&iplusjplus_r[0], &iplusjplus_r[k+i+j+2], &res_iplusjplus_r
    [0]);
  std::cout << GCL::PID << " res = " <<  res << "\n";
}

int final;
MPI_Reduce(&res, &final, 1, MPI_INT, MPI_LAND, 0, GCL::GCL_WORLD);

if (GCL::PID==0)
  if (!final) {
    std::cout << "@" << GCL::PID << "@ FAILED!\n";
  } else
    std::cout << "@" << GCL::PID << "@ PASSED!\n";


MPI_Barrier(GCL::GCL_WORLD);
```

```
  MPI_Finalize();

  return !final;
}
```