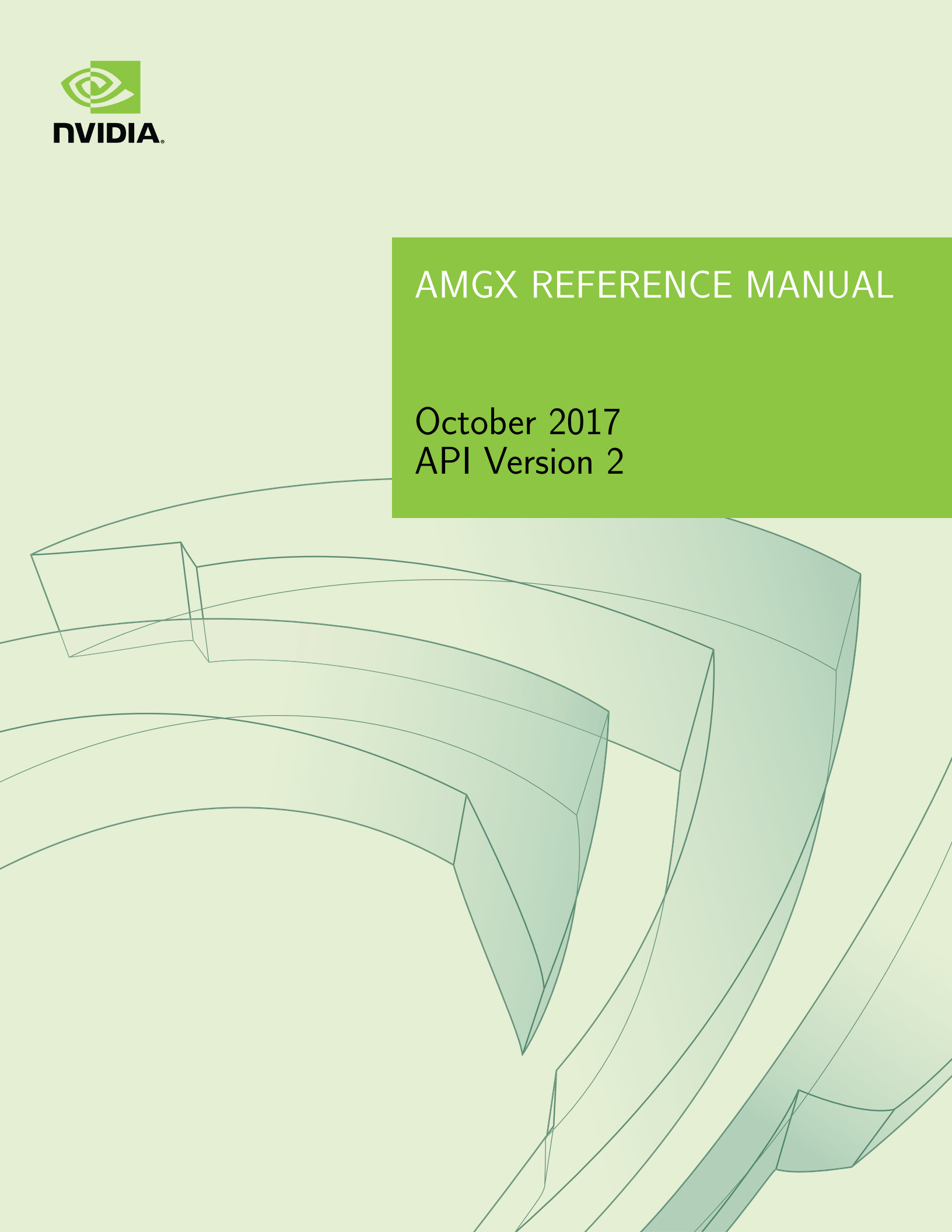

AMGX REFERENCE MANUAL

October 2017
API Version 2

Contents

1 API Reference **1**

1.1 Utilities . . . 2

1.1.1 Initialize . . . 3

1.1.2 Finalize . . . 4

1.1.3 Initialize Plugins . . . 5

1.1.4 Finalize Plugins . . . 7

1.1.5 Get API Version . . . 9

1.1.6 Get Error String . . . 10

1.1.7 Get Build Info Strings . . . 12

1.1.8 Pin Memory . . . 14

1.1.9 Unpin Memory . . . 16

1.1.10 Install Signal Handler . . . 18

1.1.11 Reset Signal Handler . . . 19

1.1.12 Register Print Callback . . . 20

1.1.13 Read System . . . 21

1.1.14 Read System Distributed . . . 24

1.1.15 Read System Global . . . 27

1.1.16 Read System Maps One Ring . . . 32

1.1.17 Free System Maps One Ring . . . 38

1.1.18 Write System . . . 41

1.1.19 Write System Distributed . . . 43

1.2 Config . . . 46

1.2.1 Config Create . . . 47

1.2.2 Config Create From File . . . 49

1.2.3 Config Get Default Number Of Rings . . . 51
1.2.4 Config Destroy . . . 53
1.3 Resources . . . 55
1.3.1 Resources Create . . . 56
1.3.2 Resources Create Simple . . . 58
1.3.3 Resources Destroy . . . 59
1.4 Solver . . . 61
1.4.1 Solver Create . . . 63
1.4.2 Solver Destroy . . . 65
1.4.3 Solver Setup . . . 67
1.4.4 Solver Solve With 0 Initial Guess . . . 69
1.4.5 Solver Solve . . . 72
1.4.6 Solver Get Iterations Number . . . 75
1.4.7 Solver Get Iteration Residual . . . 77
1.4.8 Solver Get Status . . . 79
1.5 Matrix . . . 81
1.5.1 Matrix Create . . . 83
1.5.2 Matrix Destroy . . . 85
1.5.3 Matrix Upload All . . . 87
1.5.4 Matrix Upload All Global . . . 91
1.5.5 Matrix Replace Coefficients . . . 95
1.5.6 Matrix Get Size . . . 98
1.5.7 Matrix Comm From Maps . . . 100
1.5.8 Matrix Comm From Maps One Ring . . . 104
1.6 Vector . . . 108
1.6.1 Vector Create . . . 110
1.6.2 Vector Destroy . . . 112
1.6.3 Vector Upload . . . 114
1.6.4 Vector Download . . . 116
1.6.5 Vector Set Zero . . . 118
1.6.6 Vector Get Size . . . 120
1.6.7 Vector Bind . . . 122

2 Algorithm Guide **125**

2.1 Config Syntax . . . 127
2.2 Resources Settings . . . 129
2.3 General Settings . . . 130
2.4 Multigrid Settings . . . 132
2.5 Classical Algebraic Multigrid . . . 134
2.6 Aggregation Multigrid . . . 136
2.7 Krylov Solvers . . . 138
2.8 Smoothers . . . 141
2.9 Cycles . . . 144
2.10 Examples . . . 146

3 Programming Guide **149**

3.1 Single Thread Single GPU . . . 150
3.2 Multi Thread OpenMP . . . 152
3.3 Single Thread MPI . . . 154
3.4 Multi Thread Hybrid . . . 157

4 Legacy API **159**

Chapter 1

API Reference

DESCRIPTION

This documents describes the C API for NVIDIA's AMGX library.

The API presents an object-oriented framework for describing a sparse linear system of equations and a method for solving it. There are five types of objects: **Config**, **Resources**, **Solver**, **Matrix**, and **Vector**. **Config** is a lightweight representation of a parsed set of configuration strings, the format of which is described in *Config Syntax*. **Resources** defines a set of resources, such as memory, threads, and hardware, which will be used by other objects for storage or execution. **Matrix** and **Vector** define components of a linear system, and are bound to a **Solver** object which actually executes the solution algorithm.

Utilities

Config

Resources

Solver

Matrix

Vector

HISTORY

This document describes API Version 2.

1.1 Utilities

NAME

Utilities

DESCRIPTION

This section describes utility API functions that are not specific to the core objects of the AMGX library. AMGX provides a number of mechanisms for dealing with text output, file input and output, memory management, and other application and library-level utilities.

AMGX_initialize

AMGX_finalize

AMGX_initialize_plugins

AMGX_finalize_plugins

AMGX_get_api_version

AMGX_get_error_string

AMGX_get_build_info_strings

AMGX_pin_memory

AMGX_unpin_memory

AMGX_install_signal_handler

AMGX_reset_signal_handler

AMGX_read_system

AMGX_read_system_distributed

AMGX_write_system

AMGX_write_system_distributed

AMGX_register_print_callback

HISTORY

The utility routines were updated in API version 2 to add support for distributed matrices.

SEE ALSO

Initialize, *Finalize*, *Initialize Plugins*, *Finalize Plugins*, *Get Error String*, *Get Build Info Strings*, *Pin Memory*, *Unpin Memory*, *Install Signal Handler*, *Reset Signal Handler*, *Read System*, *Read System Distributed*, *Write System*, *Register Print Callback*

1.1.1 Initialize

NAME

AMGX_initialize - Initialize the **AMGX** library.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_initialize();
```

DESCRIPTION

AMGX_initialize initializes the library. This routine must be called before any subsequent calls to AMGX routines. It is legal to call **AMGX_initialize** after a call to **AMGX_finalize**, so the at the library can be initialized and finalized repeatedly throughout the lifetime of an application. **AMGX_initialize** may perform a license check and therefore can fail with a license error.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_CORE`

`AMGX_RC_LICENSE`

`AMGX_RC_UNKNOWN`

HISTORY

AMGX_initialize was introduced in API Version 1.

SEE ALSO

Initialize Plugins, *Finalize*, *Finalize Plugins*

1.1.2 Finalize

NAME

AMGX_finalize - Finalizes the **AMGX** library.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_finalize();
```

DESCRIPTION

AMGX_finalize cleans up any resources or other the library. The caller is not allowed to call any AMGX routines after this has been called. It is legal to call **AMGX_initialize** after a call to **AMGX_finalize**, so the at the library can be initialized and finalized repeatedly throughout the lifetime of an application.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_UNKNOWN`

HISTORY

AMGX_finalize was introduced in API Version 2.

SEE ALSO

Initialize Plugins, *Initialize*, *Finalize plugins*

1.1.3 Initialize Plugins

NAME

AMGX_initialize_plugins - Initialize the built-in plugins for the **AMGX** library.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_initialize_plugins();
```

DESCRIPTION

AMGX_initialize_plugins initializes the plugins that have been compiled into the AMGX library. Since AMGX uses a plugin system for encapsulating functionality, all of the routines such as file IO, solvers, smoothers, graph coloring, and cycling strategy are implemented via built-in plugins. Therefore, attempts to call routines like **AMGX_solver_setup** or **AMGX_solver_solve** are likely to fail if the plugins have not been initialized first.

It is legal to call **AMGX_initialize_plugins** after a call to **AMGX_finalize_plugins**, so the at the plugins can be initialized and finalized repeatedly throughout the lifetime of an application.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_UNKNOWN`

EXAMPLE

```
int main(int, const char **) {
  AMGX_initialize();
  AMGX_initialize_plugins();

  // use the library

  AMGX_finalize_plugins();
  AMGX_finalize();
  return 0;
}
```

HISTORY

AMGX_initialize_plugins was introduced in API Version 1.

SEE ALSO

Initialize, *Finalize*, *Finalize Plugins*

1.1.4 Finalize Plugins

NAME

AMGX_finalize_plugins - Finalizes the built-in plugins for the **AMGX** library.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_finalize_plugins();
```

DESCRIPTION

AMGX_finalize_plugins removes all of the plugins from the AMGX library. Since AMGX uses a plugin system for encapsulating functionality, all of the routines such as file IO, solvers, smoothers, graph coloring, and cycling strategy are implemented via built-in plugins. Therefore, attempts to call routines like **AMGX_solver_setup** or **AMGX_solver_solve** are likely to fail after the plugins have been finalized.

It is legal to call **AMGX_initialize_plugins** after a call to **AMGX_finalize_plugins**, so the at the plugins can be initialized and finalized repeatedly throughout the lifetime of an application.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_UNKNOWN`

EXAMPLE

```
int main(int, const char **) {
  AMGX_initialize();
  AMGX_initialize_plugins();

  // use the library

  AMGX_finalize_plugins();
  AMGX_finalize();
  return 0;
}
```

HISTORY

AMGX_initialize_plugins was introduced in API Version 1.

SEE ALSO

Initialize, *Initialize Plugins*, *Finalize*

1.1.5 Get API Version

NAME

AMGX_get_api_version - Retrieve API version of AMGX

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_get_api_version(int *major, int *minor);
```

PARAMETERS

major

The major API version number.

minor

The minor API version number.

DESCRIPTION

AMGX_get_api_version retrieves information about the API version of AMGX. All API versions of AMGX are assigning a monotonically increasing and unique API version. It is important to check against this number to avoid compatibility issues.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

EXAMPLE

```
int major, minor;
AMGX_get_api_version(&major, &minor);
printf("AMGX API version %d.%d\n", major, minor);
```

HISTORY

AMGX_get_api_version was introduced in API Version 1.

SEE ALSO

1.1.6 Get Error String

NAME

AMGX_get_error_string - Convert an error code into a string.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_get_error_string(AMGX_RC err, char* buf, int buf_len);
```

PARAMETERS

err

The error code to be converted.

buf

The string buffer to receive the translation.

buf_len

The length of the provided string buffer. If the error string to be returned is larger than this value, only the first **buf_len** characters will be copied into **buf**.

DESCRIPTION

AMGX_get_error_string can be used to translate the error codes returned via AMGX routines into a human readable string. This is useful for debugging or diagnostics.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

EXAMPLE

```
AMGX_RC code = AMGX_initialize();
char buff[256];
AMGX_get_error_string(code, buff, 256);
printf("Initialize returned %s\n", buff);
```

HISTORY

AMGX_get_error_string was introduced in API Version 1.

SEE ALSO

1.1.7 Get Build Info Strings

NAME

AMGX_get_build_info_strings - Retrieve pointers to string containing information about this build of AMGX

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_get_build_info_strings(char **version, char **date,
  char **time);
```

PARAMETERS

version

Pointer to a `char *` that will be set to the build version. This string should not be modified.

date

Pointer to a `char *` that will be set to the build date. This string should not be modified.

time

Pointer to a `char *` that will be set to the build time. This string should not be modified.

DESCRIPTION

AMGX_get_build_info_strings retrieves information about this build of AMGX. All versions of AMGX are assigning a monotonically increasing and unique version number. This build number is important for reporting errors to the AMGX development team.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

EXAMPLE

```
char *version, *date, *time;
AMGX_get_build_info_strings(&version, &date, &time);
printf("AMGX version %s built %s %s\n", version, date, time);
```

HISTORY

AMGX_get_build_info_strings was introduced in API Version 1.

SEE ALSO

1.1.8 Pin Memory

NAME

AMGX_pin_memory - Notify the operating system that a buffer is to be pinned.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_pin_memory(void *ptr, unsigned int bytes);
```

PARAMETERS

ptr

The start of the buffer to be pinned.

bytes

The number of bytes to be pinned.

DESCRIPTION

AMGX_pin_memory notifies the operating system that a host buffer is to be pinned to reside in physical memory. This allows the CUDA driver to optimize transfers across the PCI-Express bus, resulting in higher achieved bandwidth.

It is recommended that the host buffers used in **AMGX_matrix_upload_all**, **AMGX_matrix_replace_coefficients**, **AMGX_vector_upload**, and **AMGX_vector_download** be pinned.

For details on the benefits and drawbacks of using pinned memory, please see the CUDA Programming Guide.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_UNKNOWN`

EXAMPLE

```
AMGX_vector_handle vector;
AMGX_vector_create(&vector, AMGX_mode_dDFI);
float data[] = {1, 2, 3, 4, 1, 2, 3, 4};
AMGX_pin_memory(data, sizeof(float)*8);
AMGX_vector_upload(vector, 2, 4, data);
```

HISTORY

AMGX_pin_memory was introduced in API Version 1.

SEE ALSO

Matrix Upload All, *Matrix Replace Coefficients*, *Vector Upload*, *Vector Download*, *Unpin Memory*

1.1.9 Unpin Memory

NAME

AMGX_unpin_memory - Notify the operating system that a buffer is to be unpinned.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_unpin_memory(void *ptr);
```

PARAMETERS

ptr

The buffer to be unpinned.

DESCRIPTION

AMGX_unpin_memory notifies the operating system that a host buffer which was previously pinned should be unpinned.

For details on the benefits and drawbacks of using pinned memory, please see the CUDA Programming Guide.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_UNKNOWN`

EXAMPLE

```
AMGX_vector_handle vector;
AMGX_vector_create(&vector, AMGX_mode_dDFI);
float data[] = {1, 2, 3, 4, 1, 2, 3, 4};
AMGX_pin_memory(data, sizeof(float)*8);
AMGX_vector_upload(vector, 2, 4, data);
AMGX_unpin_memory(data);
```

HISTORY

AMGX_unpin_memory was introduced in API Version 2.

SEE ALSO

Matrix Upload All, *Matrix Replace Coefficients*, *Vector Upload*, *Vector Download*, *Pin Memory*

1.1.10 Install Signal Handler

NAME

AMGX_install_signal_handler - cause AMGX to install its default signal handlers.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_install_signal_handler();
```

DESCRIPTION

AMGX_install_signal_handler causes the AMGX library to install its default signal handlers. When the signal handlers are installed, AMGX will catch fatal signals. On Windows, it will print information about the signal and exit. On Linux, it will additionally print a demangled stack trace for debugging purposes. All text output will be directed to the callback set via **AMGX_register_print_callback**.

By default, AMGX signal handlers are not installed. This allows the application to handle signals.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

HISTORY

AMGX_install_signal_handler was introduced in API Version 1.

SEE ALSO

Reset Signal Handler, *Register Print Callback*

1.1.11 Reset Signal Handler

NAME

AMGX_reset_signal_handler - cause AMGX to restore previous signal handlers.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_reset_signal_handler();
```

DESCRIPTION

AMGX_reset_signal_handler restores any signal handlers that were set prior to the last call to **AMGX_install_signal_handler**. This allows an application, for example, to enable AMGX signal handling while inside a particular AMGX routine, and restore the application's signal handler upon exiting from that routine.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

EXAMPLE

```
AMGX_install_signal_handler();
AMGX_solver_solve(solver, rhs, sol);
AMGX_reset_signal_handler();
```

HISTORY

AMGX_reset_signal_handler was introduced in API Version 1.

SEE ALSO

Install Signal Handler, *Register Print Callback*

1.1.12 Register Print Callback

NAME

AMGX_register_print_callback - Register a user-provided callback for text output

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_register_print_callback(AMGX_print_callback func);
```

PARAMETERS

func

Pointer to the callback function to register.

DESCRIPTION

AMGX_register_print_callback allows the application to register a custom text print callback which will receive all text output from the AMGX library. This allows an application to integrate AMGX messages into output or debugging logs.

By default, all text output is sent to `stdout`.

The user-provided callback should have the following signature:

```
void callback(const char *msg, int length);
```

msg will be the null-terminated string to display (possibly with newline characters embedded in it), and **length** is always guaranteed to be equal to `strlen(msg)`.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

HISTORY

AMGX_register_print_callback was introduced in API Version 1.

SEE ALSO

Reset Signal Handler, *Install Signal Handler*

1.1.13 Read System

NAME

AMGX_read_system - Read a linear system of equations from a file.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_read_system(AMGX_matrix_handle mtx,
 AMGX_vector_handle rhs, AMGX_vector_handle sol, const char *filename);
```

PARAMETERS

mtx

Handle to a **Matrix** object to be loaded from disk. The **Matrix** must already have been created via **AMGX_matrix_create**.

rhs

Handle to a **Vector** object representing the right hand size to be loaded from disk. The **Vector** must already have been created via **AMGX_vector_create**.

sol

Handle to a *Vector* object representing the solution vector to be loaded from disk. This value is optional - some file formats allow a solution vector to be stored with the matrix. If NULL is passed for this value, no solution vector data will be returned. If non-NULL, the **Vector** must already have been created via **AMGX_vector_create**. If non-NULL and the file has no associated solution vector data, the **Vector** will be initialized to have the proper number of entries, and will be set to all zeroes.

filename

Path to the file to be loaded.

DESCRIPTION

AMGX_read_system reads a linear system of equations, including the matrix, the right hand size, and an optional starting solution vector, from disk into **Matrix** and **Vector** objects. This routine should only be used in the single-threaded case, where the matrix will not be partitioned across multiple threads. For MPI execution, use **AMGX_read_system_distributed**.

The **Matrix** and **Vector** objects must have previously been created via **AMGX_matrix_create** and **AMGX_vector_create**. If a **Matrix** or **Vector** has had data uploaded to it, that data will be overwritten.

AMGX support a MatrixMarket format, with optional extensions to include the solution vector, separate storage of the diagonal, and block size information. Only 'real', 'symmetric' and 'general' keywords are currently supported. Diagonal is asumed to be stored right after the matrix, followed by right-hand-side and solution vectors. The extensions are provided through a comment line, starting with %%AMGX. Any modifier in the extensions line can be ommited and the ordering is not important. It is not assumed that

rows are sored and the column entires appear in sorted order within each row. But if this is the case, 'sorted' option can be provided to improve reading performance.

If there is no right-hand-side data in the file, **rhs** is resized and filled with 1. Similarly, if there is no solution data, **sol** is resized and filled with 0 .

The extended MatrixMarket format is as follows:

```
%%MatrixMarket matrix coordinate real general
%%AMGX block_dimx(int) block_dimy(int) diagonal sorted rhs solution
%% mxn matrix with nnz non-zero elements
%% m=block_dimx*n_block_rows, n=block_dimy*n_block_cols
%% nnz=block_dimx*block_dimy*n_block_entrees
m(int) n(int) nnz(int)
1 1 a_11
1 2 a_12
...
i j a_ij
...
%% these two comment lines present only for the description (to be removed)
%% optional diagonal mx1
...
a_ii
...
%% these two comment lines present only for the description (to be removed)
%% optional rhs mx1
...
b_i
...
%% these two comment lines present only for the description (to be removed)
%% optional solution nx1
...
x_i
...
```

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

`AMGX_RC_NO_MEMORY`

`AMGX_RC_UNKNOWN`

`AMGX_RC_IO_ERROR`

EXAMPLE

```
AMGX_resources_handle resources;
AMGX_resources_create_simple(&resources);
AMGX_matrix_handle matrix;
AMGX_matrix_create(&matrix, resources, AMGX_mode_dFFI);
AMGX_vector_handle rhs;
AMGX_vector_create(&rhs, resources, AMGX_mode_dFFI);
AMGX_read_system(matrix, vector, NULL, argv[1]);
```

Description of a matrix with 32 block rows and block columns, 100 block entrees, and block size of 4x4:

```
%%MatrixMarket matrix coordinate real general
%%AMGX 4 4
128 128 1600
%% these two comment lines present only for the description (to be removed)
%% i = 0..31, j = 0..31
4*i+1 4*j+1 value
4*i+1 4*j+2 value
...
4*i+2 4*j+1 value
...
4*i+4 4*j+4 value
...
```

HISTORY

AMGX_read_system was introduced in API Version 1, the calling signature was modified in API Version 2.

SEE ALSO

Write_system, *Matrix Create*, *Vector Create*, *Read System Distributed*

1.1.14 Read System Distributed

NAME

AMGX_read_system_distributed - Read a subset of linear system of equations from a file in a distributed application.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_read_system_distributed(AMGX_matrix_handle mtx,
 AMGX_vector_handle rhs, AMGX_vector_handle sol,
 const char *filename, int allocated_halo_depth, int num_partitions,
 int *partition_sizes, int partition_vector_size, int *partition_vector);
```

PARAMETERS

mtx

Handle to a **Matrix** object to be loaded from disk. The **Matrix** must already have been created via **AMGX_matrix_create**.

rhs

Handle to a **Vector** object representing the right hand size to be loaded from disk. The **Vector** must already have been created via **AMGX_vector_create**.

sol

Handle to a *Vector* object representing the solution vector to be loaded from disk. This value is optional - some file formats allow a solution vector to be stored with the matrix. If NULL is passed for this value, no solution vector data will be returned. If non-NULL, the **Vector** must already have been created via **AMGX_vector_create**. If non-NULL and the file has no associated solution vector data, the **Vector** will be initialized to have the proper number of entries, and will be set to all zeroes.

filename

Path to the file to be loaded.

allocated_halo_depth

In order to support halo exchanges for a given halo depth, the **Matrix** must allocate enough memory to store any extra layers of data from remote partitions. This setting causes the **Matrix** to allocate enough memory to support halo exchanges for halo depth of **allocated_halo_depth**. This should be at least as large as the depth of the halo region to be sent to neighboring MPI ranks, but not larger than necessary, since larger values result in more memory being allocated and more overhead during the communication map construction. Currently, **allocated_halo_depth** must be equal to **num_import_rings**. In the future, AMGX may support computing extra layers of halo regions automatically.

num_partitions

The total number of partitions. Typically, this will match the number of MPI ranks or threads.

partition_sizes

An array of size **num_partitions** listing the size of each partition. That is, `partition_sizes[i]` will be the number of block-rows in the system matrix in partition `i`.

If NULL is passed for this value, partition sizes are computed from partition vector.

partition_vector_size

The number of entries in the **partition_vector** array. This must be equal to the number of block-rows of the global system matrix, which is being read for disk. If these sizes do not match, it will result in an error.

partition_vector

An array of partition assignments of the global system matrix. The array must have size equal to **partition_vector_sizes**, which is equal to the number of block-rows of the global system matrix. Each entry `partition_vector[i]` will be an integer between 0 and `num_partitions-1` indicating the partition to which block-row `i` belongs. The total number of entries in **partition_vector** with values equal to `j` must be equal to the value specified in `partition_sizes[j]`.

The partitioning is typically obtained via some type of mesh partitioner and therefore this information is assumed to be avialable to the calling application, perhaps stored on disk seperately from the global system matrix.

If NULL is passed for this value, trivial partitioning is performed, when block rows are evenly distributed among different ranks (block rows `0..k` go to rank 0, etc)

DESCRIPTION

AMGX_read_system_distributed reads a linear system of equations, including the matrix, the right hand size, and an optional starting solution vector, from disk into **Matrix** and **Vector** objects. Unlike **AMGX_read_system**, it takes partitioning information and only reads the portion of the system matrix which is indicated to belong to this MPI rank. This routine is optimized to only store data relevant to the local partition, and therefore may be used to load data from a matrix which is too large to fit into system memory.

All objects must have previously been created via **AMGX_matrix_create** and **AMGX_vector_create**. If a **Matrix** or **Vector** object has had data uploaded to them, that data will be overwritten.

After this routine, there is no need to call **AMGX_matrix_comm_from_maps**, **AMGX_matrix_comm_from_maps_one_ring**, or **AMGX_vector_bind** on the **Matrix** or **Vector** objects since the reader will establish communication maps internally.

See *Read System* for information about the supported file format.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

`AMGX_RC_NO_MEMORY`

AMGX_RC_UNKNOWN

AMGX_RC_IO_ERROR

EXAMPLE

```
AMGX_config_handle config;
AMGX_config_create(&config, ""); // use default options
AMGX_resources_handle resources;
int gpu_ids[] = {0};
AMGX_resources_create(&resources, config, MPI_COMM_WORLD, 1, gpu_ids);
AMGX_matrix_handle matrix;
AMGX_matrix_create(&matrix, resources, AMGX_mode_dFFI);
AMGX_vector_handle rhs;
AMGX_vector_create(&rhs, resources, AMGX_mode_dFFI);
// assume a 4x4 matrix, where rows 0 and 2 belong to partition 0, 1 and 3 to partition 1.
int partition_vector[] = {0,1,0,1};
int partition_sizes[] = {2,2};
AMGX_read_system_distributed(matrix, vector, 0, argv[1], 1, 2, partition_sizes, 4, partition_vector);
```

HISTORY

AMGX_read_system_distributed was introduced in API version 2.

SEE ALSO

Matrix Create, *Vector Create*, *Read System*, *Write System*, *Matrix Upload All*, *Matrix Comm From Maps*, *Matrix Comm From Maps One Ring*, *Vector Bind*

1.1.15 Read System Global

NAME

AMGX_read_system_global - Read a subset of linear system of equations from a file in a distributed application to C buffers (using global column indices), allocated internally.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_read_system_global(int *n, int *nnz, int *block_dimx,
 int *block_dimy, int **row_ptrs, void **col_indices_global, void **data, void **diag_data,
 void **rhs, void **sol, AMGX_resources_handle rsrc, AMGX_Mode mode, const char *filename,
 int allocated_halo_depth, int num_partitions, const int *partition_sizes,
 int partition_vector_size, const int *partition_vector);
```

PARAMETERS

n

The dimension of the local matrix in terms of block-units. This also corresponds to the number of rows (in block-units) in this partition.

nnz

The number of non-zero entries in the local CSR matrix, in terms of block-units.

block_dimx

The blocksize in x direction. For a scalar matrix, this value should be 1. Currently only square blocks are supported, so **block_dimx** and **block_dimy** must be equal. Currently only **block_dimx**=1 is supported.

block_dimy

The blocksize in y direction. For a scalar matrix, this value should be 1. Currently only square blocks are supported, so **block_dimx** and **block_dimy** must be equal. Currently only **block_dimy**=1 is supported.

row_ptrs

Array of indices into the **col_indices_global** structure. **row_ptrs** has `n+1` entries. Entry `i` indicates the starting index of the values belonging to row `i` in the **col_indices_global** table. `row_ptrs[0]` must always be 0, and `row_ptrs[n]` must always be equal to **nnz**.

col_indices_global

Array of the `global` column indices of the nonzero blocks in the matrix. The type of column indices must be 64-bit integer (int64_t). **col_indices_global** must have **nnz** entries. `col_indices_global[i]` contains the column index (in block-units) of nonzero block `i`.

data

Array of the matrix entries in "array of structures" (AoS) layout. **data** must have `nnz * block_dimx * block_dimy` entries, where **block_dimx** and **block_dimy** are, for example, both 4 in the case of a 4x4 block matrix, or 1 in the case of a scalar matrix. `data[i*block_dimx*block_dimy]` contains the

entry 0,0 in block `i`. `data[i*block_dimx*block_dimy+1*block_dimy]` contains the entry 1,0 in block `i`, and so on. Data within the block is assumed to be arranged in row-major scanline order.

diag_data

A pointer to an array containing external diagonal entries for each row. If diag property does not exist, the pointer will be set to NULL. If diag property exists, the array will be allocated and filled. In this case, it is assumed to be an array with `n*block_dimx*block_dimy` entries in AoS layout, where `diag_data[i*block_dimx*block_dimy]` is the 0,0 entry in the i,i block in the matrix. Currently external diagonal is not supported.

rhs

Handle to a **Vector** object representing the right hand size to be loaded from disk. The **Vector** must already have been created via **AMGX_vector_create**.

sol

Handle to a *Vector* object representing the solution vector to be loaded from disk. This value is optional - some file formats allow a solution vector to be stored with the matrix. If NULL is passed for this value, no solution vector data will be returned. If non-NULL, the **Vector** must already have been created via **AMGX_vector_create**. If non-NULL and the file has no associated solution vector data, the **Vector** will be initialized to have the proper number of entries, and will be set to all zeroes.

rsrc

The **Resources** object which defines where the memory associated with this object will be allocated, the precision of this vector object, and any information about how it will communicate with other vectors in other MPI ranks or threads.

mode

The **mode** parameter contains the information indicating:

1. (lowercase) letter: whether the code will run on the host (h) or device (d).
2. (uppercase) letter: whether the matrix precision is float (F) or double (D).
3. (uppercase) letter: whether the vector precision is float (F) or double (D).
4. (uppercase) letter: whether the index type is 32-bit int (I) or else (not currently supported).

The corresponding enum combinations are listed below:

```
typedef enum {
  AMGX_mode_hDDI, // 8192
  AMGX_mode_hDFI, // 8448
  AMGX_mode_hFFI, // 8464
  AMGX_mode_dDDI, // 8193
  AMGX_mode_dDFI, // 8449
  AMGX_mode_dFFI  // 8465
} AMGX_Mode;
```

Note that AMGX does not currently perform automatic precision conversion, so the data that is passed into a **Vector** object via subsequent calls to **AMGX_vector_upload** and **AMGX_vector_download** must match the precision of the **mode** parameter when the **Vector** was created.

filename

Path to the file to be loaded.

allocated_halo_depth

In order to support halo exchanges with overlap, it is necessary to allocate buffers to store halo vertices and connectivity data. The **allocated_halo_depth** setting causes the **Matrix** to allocate enough storage to support halo exchanges for halos up to the specified depth. The actual amount of overlap is specified via a setting in the **Resources** and **Solver** object. Currently, **allocated_halo_depth** must be equal to **num_import_rings**. In the future, AMGX may support computing extra layers of halo regions automatically.

num_partitions

The total number of partitions. Typically, this will match the number of MPI ranks or threads.

partition_sizes

An array of size **num_partitions** listing the size of each partition. That is, `partition_sizes[i]` will be the number of block-rows in the system matrix in partition `i`.

If NULL is passed for this value, partition sizes are computed from partition vector.

partition_vector_size

The number of entries in the **partition_vector** array. This must be equal to the number of block-rows of the global system matrix, which is being read for disk. If these sizes do not match, it will result in an error.

partition_vector

An array of partition assignments of the global system matrix. The array must have size equal to **partition_vector_sizes**, which is equal to the number of block-rows of the global system matrix. Each entry `partition_vector[i]` will be an integer between 0 and `num_partitions-1` indicating the partition to which block-row `i` belongs. The total number of entries in **partition_vector** with values equal to `j` must be equal to the value specified in `partition_sizes[j]`.

The partitioning is typically obtained via some type of mesh partitioner and therefore this information is assumed to be avialable to the calling application, perhaps stored on disk seperately from the global system matrix.

If NULL is passed for this value, trivial partitioning is performed, when block rows are evenly distributed among different ranks (block rows `0..k` go to rank 0, etc)

DESCRIPTION

AMGX_read_system_global reads a linear system of equations, including the matrix, the right hand size, and an optional starting solution vector, from disk into to C buffers, allocated internally. See *Read System* for information about the supported file format.

Unlike **AMGX_read_system**, it takes partitioning information and only reads the portion of the system matrix which is indicated to belong to this MPI rank. This routine is optimized to only store data relevant to the local partition, and therefore may be used to load data from a matrix which is too large to fit into system memory. It does not perform packing of the local matrix.

In distributed setting, the local matrix (with global column indices) is obtained from the global coefficient matrix on rank `i` following these steps:

1. Select the rows (with global column indices) belonging to rank `i` and read them into memory

2. The resulting local matrix (with global column indices) is passed to routine **AMGX_matrix_upload_all_global**.

3. The resulting local right-hand-side and initial-guess vectors are passed to **AMGX_vector_bind** and **AMGX_vector_upload**.

All output arrays are allocated inside of the function, only pointers for the relevant data need to be provided.

RETURN VALUES

Relevant return values:

AMGX_RC_OK

AMGX_RC_BAD_PARAMETERS

AMGX_RC_BAD_MODE

AMGX_RC_NO_MEMORY

AMGX_RC_UNKNOWN

AMGX_RC_IO_ERROR

EXAMPLE

```
AMGX_config_handle config;
AMGX_resources_handle rsrc;
AMGX_matrix_handle matrix;
int gpu_ids[] = {0};
int n, nnz, block_dimx, block_dimy, num_neighbors;
int *row_ptrs = NULL, *col_indices_global = NULL,
void *values = NULL, *diag = NULL, *x_data = NULL, *b_data = NULL;


AMGX_config_create(&config, ""); // use default options
AMGX_resources_create(&rsrc, config, MPI_COMM_WORLD, 1, gpu_ids);
AMGX_matrix_create(&matrix, rsrc, AMGX_mode_dFFI);


// assume a 4x4 matrix, where rows 0 and 2 belong to partition 0, 1 and 3 to partition 1.
int partition_vector[] = {0,1,0,1};
int partition_sizes[] = {2,2};


// partition sizes and number of partitions will be computed from partition vector
AMGX_read_system_global(&n, &nnz, &block_dimx, &block_dimy, &row_ptrs,
  &col_indices_global, &values, &diag, &b_data, &x_data,
  rsrc, AMGX_mode_dFFI, argv[1], 1, 2, partition_sizes, 4, partition_vector);


AMGX_matrix_upload_all_global(A, 4, n, nnz, block_dimx, block_dimy, row_ptrs,
  col_indices_global, values, diag, 1 /*or 2*/, 1 /*or 2*/, partition_vector);


...
```

HISTORY

AMGX_read_system_global was introduced in API version 2.

SEE ALSO

Matrix Create, *Vector Create*, *Read System*, *Write System*, *Matrix Comm From Maps*, *Matrix Comm From Maps One Ring*, *Matrix Upload All Global*, *Vector Bind*

1.1.16 Read System Maps One Ring

NAME

AMGX_read_system_maps_one_ring - Read a subset of linear system of equations from a file in a distributed application to C buffers (using local column indices), allocated internally.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_read_system_maps_one_ring(int *n, int *nnz, int *block_dimx,
 int *block_dimy, int **row_ptrs, int **col_indices, void **data, void **diag_data,
 void **rhs, void **sol, int *num_neighbors, int **neighbors, int **send_sizes,
 int ***send_maps, int **recv_sizes, int ***recv_maps,
 AMGX_resources_handle rsrc, AMGX_Mode mode, const char *filename,
 int allocated_halo_depth, int num_partitions, const int *partition_sizes,
 int partition_vector_size, const int *partition_vector);
```

PARAMETERS

n

Pointer to an int which will be set to the matrix dimension in block-units. AMGX only allows square matrices, so this is both the number of columns and rows. In multithreaded execution, where column indices may index entries larger than the number of rows, this will return the number of rows in the local matrix partition.

nnz

Pointer to an int which will be set to the number of non-zero elements in block-units.

block_dimx

Pointer to an int which will be set to the size in x of a matrix block.

block_dimy

Pointer to an int which will be set to the size in y of a matrix block.

row_ptrs

Pointer to an array, which will be allocated and filled with indices into the col_indices structure. **row_ptrs** has `n+1` entries. Entry `i` indicates the starting index of the values belonging to row `i` in the **col_indices** table. `row_ptrs[0]` will always be 0, and `row_ptrs[n]` will always be equal to **nnz**.

col_indices

Pointer to an array, which will be allocated and filled with the column indices of the nonzero blocks in the matrix. **col_indices** will have **nnz** entries. `col_indices[i]` contains the column index (in block units) of nonzero block `i`.

data

Pointer to an array, which will be allocated and filled with the matrix entries in "array of structures" (AoS) layout. **data** will have `nnz * block_dimx * block_dimy` entries, where **block_dimx** and **block_dimy** are, for example, both 4 in the case of a 4x4 block matrix, or 1 in the case of a

scalar matrix. `data[i*block_dimx*block_dimy]` contains the entry 0,0 in block `i`. `data[i*block_dimx*block_dimy+1]` contains the entry 1,0 in block `i`, and so on. Data within the block is arranged in row-major scanline order.

diag_data

A pointer to an array containing external diagonal entries for each row. If diag property does not exist, the pointer will be set to NULL. If diag property exists, the array will be allocated and filled. In this case, it is assumed to be an array with `n*block_dimx*block_dimy` entries in AoS layout, where `diag_data[i*block_dimx*block_dimy]` is the 0,0 entry in the i,i block in the matrix.

rhs

Pointer to an array, which will be allocated and filled with the right-hand-side to be loaded from disk. If the right-hand-side is not available, the pointer will be set to NULL.

sol

Pointer to an array, which will be allocated and filled with the solution vector to be loaded from disk. If the solution vector is not available, the pointer will be set to NULL.

num_neighbors

Pointer to an int which will be set to the number of MPI ranks which share a boundary with this rank. In other words, the number of MPI ranks which will exchange data via halo exchanges.

neighbors

Pointer to an int array, which will be allocated to have size **num_neighbors** and filled, listing the index of each MPI rank that shares a boundary with this rank. In other words, a list of MPI ranks which will exchange data via halo exchanges.

send_sizes

Pointer to an int array, which will be allocated to have size **num_neighbors** and filled. The value in entry `i` is the number of local (i.e. non-halo) rows in this rank's matrix partition which will be sent to the MPI rank `neighbors[i]`.

send_maps

Pointer to an int* array, which will be allocated to contain **num_neighbors** pointers to int arrays, where entry `i` is an allocated int array of size `send_sizes[i]`. Array `i` is a "map" specifying the local row indices from this matrix partition which will be sent to the MPI rank `neighbors[i]`. The data corresponding to these local row indices will be packed into a transfer buffer, and then sent to the corresponding MPI rank. The order in which the local row indices are listed corresponds to the order in which they will be packed into the transfer buffer. For simple cases, this will be an offset mapping, where local index `n+j` (which is a halo index) will map to position `j` in the transfer buffer.

recv_sizes

Pointer to an int array, which will be allocated to have size **num_neighbors** and filled. The value in entry `i` is the number of non-local (i.e. halo) rows in this rank's matrix partition which will be received from the MPI rank `neighbors[i]`.

recv_maps

Pointer to an int* array, which will be allocated to contain **num_neighbors** pointers to int arrays, where entry `i` is an allocated int array of size `recv_sizes[i]`. Array `i` is a "map" specifying the local halo indices from this matrix partition which will be received from the MPI rank `neighbors[i]`. The data received from `neighbor[i]` will have been packed into a transfer buffer in the order specified by that remote matrix partition's **send_maps** value corresponding to this (local) MPI rank. The order in which the indices appear in **send_maps** must therefore correspond to this order.

rsrc

The **Resources** object which defines where the memory associated with this object will be allocated, the precision of this vector object, and any information about how it will communicate with other vectors in other MPI ranks or threads.

mode

The **mode** parameter contains the information indicating:

1. (lowercase) letter: whether the code will run on the host (h) or device (d).
2. (uppercase) letter: whether the matrix precision is float (F) or double (D).
3. (uppercase) letter: whether the vector precision is float (F) or double (D).
4. (uppercase) letter: whether the index type is 32-bit int (I) or else (not currently supported).

The corresponding enum combinations are listed below:

```
typedef enum {
  AMGX_mode_hDDI, // 8192
  AMGX_mode_hDFI, // 8448
  AMGX_mode_hFFI, // 8464
  AMGX_mode_dDDI, // 8193
  AMGX_mode_dDFI, // 8449
  AMGX_mode_dFFI  // 8465
} AMGX_Mode;
```

Note that AMGX does not currently perform automatic precision conversion, so the data that is passed into a **Vector** object via subsequent calls to **AMGX_vector_upload** and **AMGX_vector_download** must match the precision of the **mode** parameter when the **Vector** was created.

filename

Path to the file to be loaded.

allocated_halo_depth

In order to support halo exchanges for a given halo depth, the **Matrix** must allocate enough memory to store any extra layers of data from remote partitions. This setting causes the **Matrix** to allocate enough memory to support halo exchanges for halo depth of **allocated_halo_depth**. This should be at least as large as the depth of the halo region to be sent to neighboring MPI ranks, but not larger than necessary, since larger values result in more memory being allocated and more overhead during the communication map construction. Currently, **allocated_halo_depth** must be equal to `num_import_rings` and therefore here it equals 1 (which is implicit from the name of the routine). In the future, AMGX may support deeper halo regions.

num_partitions

The total number of partitions. Typically, this will match the number of MPI ranks or threads.

partition_sizes

An array of size **num_partitions** listing the size of each partition. That is, `partition_sizes[i]` will be the number of block-rows in the system matrix in partition `i`.

If NULL is passed for this value, partition sizes are computed from partition vector.

partition_vector_size

The number of entries in the **partition_vector** array. This must be equal to the number of block-rows of the global system matrix, which is being read for disk. If these sizes do not match, it will result in an error.

partition_vector

An array of partition assignments of the global system matrix. The array must have size equal to **partition_vector_sizes**, which is equal to the number of block-rows of the global system matrix. Each entry `partition_vector[i]` will be an integer between 0 and `num_partitions-1` indicating the partition to which block-row `i` belongs. The total number of entries in **partition_vector** with values equal to `j` must be equal to the value specified in `partition_sizes[j]`.

The partitioning is typically obtained via some type of mesh partitioner and therefore this information is assumed to be avialable to the calling application, perhaps stored on disk seperately from the global system matrix.

If NULL is passed for this value, trivial partitioning is performed, when block rows are evenly distributed among different ranks (block rows `0..k` go to rank 0, etc)

DESCRIPTION

AMGX_read_system_maps_one_ring reads a linear system of equations, including the matrix, the right hand size, and an optional starting solution vector, from disk into to C buffers, allocated internally. See *Read System* for information about the supported file format.

Unlike **AMGX_read_system**, it takes partitioning information and only reads the portion of the system matrix which is indicated to belong to this MPI rank. This routine is optimized to only store data relevant to the local partition, and therefore may be used to load data from a matrix which is too large to fit into system memory. It also performs packing of the local matrix as described below.

In distributed setting, the packed local matrix is obtained from the global coefficient matrix on rank `i` following these steps:

1. Select the rows (with global column indices) belonging to rank `i` and read them into memory

2. Pack the columns:

a) Reorder the column indices such that the columns corresponding to the global diagonal elements present on rank `i` come first.

b) Then, reorder the remaining column indices, in the order to which they belong (have connections) to rank's `i` neighbors. Columns belonging to neighbor 0 first, neighbor 1 second, etc., leaving the natural ordering of columns within the same neighbor.

c) Notice that we now have a rectangular matrix with reordered column indices that are numbered locally. To keep track of this reordering create a local_to_global map which indicates how the locally renumbered columns map into their global counterparts.

3. Reorder the rows and columns (such that the rows with no connections to neighbors come first):

a) Find the reordering that moves rows with no connections to neighbors first

b) Apply it to rows and columns (belonging to the global diagonal elements present on rank `i` only, in other words, columns up to **n**).

4. The resulting packed local matrix is passed to routine **AMGX_matrix_upload_all**.

5. The resulting correspondingly reordered local right-hand-side and initial-guess vectors are passed to **AMGX_vector_bind** and **AMGX_vector_upload**.

This routine also computes all the required communication maps internally and exposes them for later use in a subsequent call to **AMGX_matrix_comm_from_maps_one_ring**.

All output arrays are allocated inside of the function, only pointers for the relevant data need to be provided. **AMGX_free_system_maps_one_ring** is used for later cleanup.

RETURN VALUES

Relevant return values:

AMGX_RC_OK

AMGX_RC_BAD_PARAMETERS

AMGX_RC_BAD_MODE

AMGX_RC_NO_MEMORY

AMGX_RC_UNKNOWN

AMGX_RC_IO_ERROR

EXAMPLE

```
AMGX_config_handle config;
AMGX_resources_handle resources;
AMGX_matrix_handle matrix;
int gpu_ids[] = {0};
int n, nnz, block_dimx, block_dimy, num_neighbors;
int *row_ptrs = NULL, *col_indices = NULL, *neighbors = NULL;
void *values = NULL, *diag = NULL, *x_data = NULL, *b_data = NULL;
int **send_maps = NULL;
int **recv_maps = NULL;
int *send_sizes = NULL;
int *recv_sizes = NULL;


AMGX_config_create(&config, ""); // use default options
AMGX_resources_create(&resources, config, MPI_COMM_WORLD, 1, gpu_ids);
AMGX_matrix_create(&matrix, rsrc, AMGX_mode_dFFI);


// assume a 4x4 matrix, where rows 0 and 2 belong to partition 0, 1 and 3 to partition 1.
int partition_vector[] = {0,1,0,1};


// partition sizes and number of partitions will be computed from partition vector
AMGX_read_system_maps_one_ring(&n, &nnz, &block_dimx, &block_dimy, &row_ptrs,
  &col_indices, &values, &diag, &b_data, &x_data,
  &num_neighbors, &neighbors, &send_sizes, &send_maps, &recv_sizes, &recv_maps,
  rsrc, AMGX_mode_dDDI, argv[1], 1, 0, NULL, 4, partition_vector);
```

```
AMGX_matrix_comm_from_maps_one_ring(matrix, 1, num_neighbors, neighbors,
  send_sizes, (const int **)send_maps, recv_sizes, (const int **)recv_maps);

AMGX_matrix_upload_all(matrix, n, nnz, block_dimx, block_dimy,
  row_ptrs, col_indices, values, diag);

...

AMGX_free_system_maps_one_ring(row_ptrs, col_indices, values, diag, b_data,
 x_data, num_neighbors, neighbors, send_sizes, send_maps, recv_sizes, recv_maps);

...
```

HISTORY

AMGX_read_system_maps_one_ring was introduced in API version 2.

SEE ALSO

Matrix Create, *Vector Create*, *Read System*, *Write System*, *Read System Distributed*, *Free System Maps One Ring*, *Matrix Upload All*, *Matrix Comm From Maps*, *Matrix Comm From Maps One Ring*, *Vector Bind*, *Vector Upload*

1.1.17 Free System Maps One Ring

NAME

AMGX_free_system_maps_one_ring - Frees buffers, allocated by **AMGX_read_system_maps_one_ring**.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_free_system_maps_one_ring(int *row_ptrs, int *col_indices, void *data,
 void *diag_data, void *rhs, void *sol, int num_neighbors, int *neighbors,
 int *send_sizes, int **send_maps, int *recv_sizes, int **recv_maps);
```

PARAMETERS

row_ptrs

Array of the column index of the nonzero blocks in the matrix. **col_indices** must have **nnz** entries. `col_indices[i]` contains the column index (in block units) of nonzero block `i`. If the data is not available, the pointer should be set to NULL.

col_indices

Array of the column index of the nonzero blocks in the matrix. **col_indices** must have **nnz** entries. `col_indices[i]` contains the column index (in block units) of nonzero block `i`. If the data is not available, the pointer should be set to NULL.

data

Array of the matrix entries in "array of structures" (AoS) layout. **data** must have `nnz * block_dimx * block_dimy` entries, where **block_dimx** and **block_dimy** are, for example, both 4 in the case of a 4x4 block matrix, or 1 in the case of a scalar matrix. `data[i*block_dimx*block_dimy]` contains the entry 0,0 in block `i`. `data[i*block_dimx*block_dimy+1]` contains the entry 1,0 in block `i`, and so on. Data within the block is assumed to be arranged in row-major scanline order. If the data is not available, the pointer should be set to NULL.

diag_data

Optional array of external diagonal entries for each row. If there is no external diagonal, in other words, the diagonal is contained in the matrix itself the diag_data must be set to a null pointer. If this value is non-null, it is assumed to be an array with `n*block_dimx*block_dimy` entries in AoS layout. `diag_data[i*block_dimx*block_dimy]` is the 0,0 entry in the i,i block in the matrix.

rhs

Array with right-hand-side data. If the right-hand-side data is not available, the pointer should be set to NULL.

sol

Array with solution vector. If the solution vector is not available, the pointer should be set to NULL.

num_neighbors

Number of MPI ranks which share a boundary with this rank. In other words, the number of MPI ranks which will exchange data via halo exchanges.

neighbors

An array of size **num_neighbors** listing the index of each MPI rank that shares a boundary with this rank. In other words, a list of MPI ranks which will exchange data via halo exchanges. If the data is not available, the pointer should be set to NULL.

send_sizes

An array of size **num_neighbors**. The value in entry `i` is the number of local (i.e. non-halo) rows in this rank's matrix partition which will be sent to the MPI rank `neighbors[i]`. If the data is not available, the pointer should be set to NULL.

send_maps

An array of size **num_neighbors** of arrays, where entry `i` is another array of size `send_sizes[i]`. Array `i` is a "map" specifying the local row indices from this matrix partition which will be sent to the MPI rank `neighbors[i]`. The data corresponding to these local row indices will be packed into a transfer buffer, and then sent to the corresponding MPI rank. The order in which the local row indices are listed corresponds to the order in which they will be packed into the transfer buffer. For simple cases with `n` local rows, this will be an offset mapping, where local index `n+j` (which is a halo index) will map to position `j` in the transfer buffer. If the data is not available, the pointer should be set to NULL.

recv_sizes

An array of size **num_neighbors**. The value in entry `i` is the number of non-local (i.e. halo) rows in this rank's matrix partition which will be received from the MPI rank `neighbors[i]`. If the data is not available, the pointer should be set to NULL.

recv_maps

An arry of size **num_neighbors** of arrays, where entry `i` is another array of size `recv_sizes[i]`. Array `i` is a "map" specifying the local halo indices from this matrix partition which will be received from the MPI rank `neighbors[i]`. The data received from `neighbor[i]` will have been packed into a transfer buffer in the order specified by that remote matrix partition's **send_maps** value corresponding to this (local) MPI rank. The order in which the indices appear in **send_maps** must therefore correspond to this order. If the data is not available, the pointer should be set to NULL.

DESCRIPTION

AMGX_free_system_maps_one_ring frees memory buffers, allocated by **AMGX_read_system_maps_one_ring**.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_MODE`

`AMGX_RC_UNKNOWN`

EXAMPLE

```
AMGX_config_handle config;
AMGX_config_create(&config, ""); // use default options
AMGX_resources_handle resources;
int gpu_ids[] = {0};
AMGX_resources_create(&resources, config, MPI_COMM_WORLD, 1, gpu_ids);
int n, nnz, block_dimx, block_dimy, num_neighbors;
int *row_ptrs = NULL, *col_indices = NULL, *neighbors = NULL;
void *values = NULL, *diag = NULL, *x_data = NULL, *b_data = NULL;
int **send_maps = NULL;
int **recv_maps = NULL;
int *send_sizes = NULL;
int *recv_maps_sizes = NULL;
// assume a 4x4 matrix, where rows 0 and 2 belong to partition 0, 1 and 3 to partition 1.
int partition_vector[] = {0,1,0,1};
// partition sizes and number of partitions will be computed from partition vector
AMGX_read_system_maps_one_ring(&n, &nnz, &block_dimx, &block_dimy, &row_ptrs,
 &col_indices, &values, &diag, &b_data, &x_data,
 &num_neighbors, &neighbors, &send_sizes, &send_maps, &recv_sizes, &recv_maps,
 rsrc, AMGX_mode_dDDI, argv[1], 1, 0, NULL, partition_vector_size, partition_vector);
// ...
// free allocated memory
AMGX_free_system_maps_one_ring(row_ptrs, col_indices, values, diag, b_data,
 x_data, num_neighbors, neighbors, send_sizes, send_maps, recv_sizes, recv_maps);
```

HISTORY

AMGX_free_system_maps_one_ring was introduced in API version 2.

SEE ALSO

Matrix Create, *Vector Create*, *Read System*, *Write System*, *Read System Distributed*, *Read System Maps One Ring*, *Matrix Comm From Maps*, *Matrix Comm From Maps One Ring*, *Vector Bind*

1.1.18 Write System

NAME

AMGX_write_system - Write a linear system of equations to a file in Matrix Market format.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_write_system(const AMGX_matrix_handle mtx,
 const AMGX_vector_handle rhs, const char *filename);
```

PARAMETERS

mtx

Handle to a **Matrix** object to be written to disk.

rhs

Handle to a **Vector** object representing the right hand size to be written to disk.

filename

Path to the file to be written.

DESCRIPTION

AMGX_read_system writes a linear system of equations, including both the matrix and the right hand size, to disk in Matrix Market format.

In the future, additional file formats may be supported.

RETURN VALUES

Relevant return values:

AMGX_RC_OK

AMGX_RC_BAD_PARAMETERS

AMGX_RC_BAD_MODE

AMGX_RC_UNKNOWN

AMGX_RC_IO_ERROR

EXAMPLE

```
// define a scalar identity matrix and the zero vector
// 1 0 0 0
```

```
// 0 1 0 0
// 0 0 1 0
// 0 0 0 1
float data[] = {1, 1, 1, 1};
int col_ind[] = {0, 1, 2, 3};
int row_ptr[] = {0, 1, 2, 3, 4};

AMGX_resources_handle rsrc;
AMGX_resources_create_simple(rsrc);

AMGX_matrix_handle matrix;
AMGX_matrix_create(&matrix, rsrc, AMGX_mode_dFFI);
AMGX_matrix_upload_all(matrix, 2, 2, 2, 2, row_ptr, col_ind, data, 0);

AMGX_vector_handle vector;
NVMAG_vector_create(&vector, rsrc, AMGX_mode_dFFI);
AMGX_vector_set_zero(vector, 4, 1);

AMGX_write_system(matrix, vector, "identity_system.mtx");
```

HISTORY

AMGX_write_system was introduced in API Version 1.

SEE ALSO

Read System, *Read System Distributed*

1.1.19 Write System Distributed

NAME

AMGX_write_system_distributed - Gather a linear system of equations to a single node and write it into a single file in Matrix Market format.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_write_system_distributed(const AMGX_matrix_handle mtx,
 const AMGX_vector_handle rhs, const AMGX_vector_handle sol, int allocated_halo_depth,
                                                int num_partitions,
                                                const int *partition_sizes,
                                                int partition_vector_size,
                                                const int *partition_vector,
                                                AMGX_ERROR& rc);
```

PARAMETERS

mtx

Handle to a **Matrix** object to be written to disk.

rhs

Handle to a **Vector** object representing the right hand size to be written to disk.

sol

Handle to a *Vector* object representing the solution vector to be written to disk.

filename

Path to the file to be written.

allocated_halo_depth

In order to support halo exchanges for a given halo depth, the **Matrix** must allocate enough memory to store any extra layers of data from remote partitions. This setting causes the **Matrix** to allocate enough memory to support halo exchanges for halo depth of **allocated_halo_depth**. This should be at least as large as the depth of the halo region to be sent to neighboring MPI ranks, but not larger than necessary, since larger values result in more memory being allocated and more overhead during the communication map construction. Currently, **allocated_halo_depth** must be equal to **num_import_rings**. In the future, AMGX may support computing extra layers of halo regions automatically.

num_partitions

The total number of partitions. Typically, this will match the number of MPI ranks or threads.

partition_sizes

An array of size **num_partitions** listing the size of each partition. That is, `partition_sizes[i]` will be the number of block-rows in the system matrix in partition `i`.

If NULL was passed for this value when the system was read/loaded, partition sizes were computed from partition vector.

partition_vector_size

The number of entries in the **partition_vector** array. This must be equal to the number of block-rows of the global system matrix, which is being read for disk. If these sizes do not match, it will result in an error.

partition_vector

An array of partition assignments of the global system matrix. The array must have size equal to **partition_vector_sizes**, which is equal to the number of block-rows of the global system matrix. Each entry `partition_vector[i]` will be an integer between 0 and `num_partitions-1` indicating the partition to which block-row `i` belongs. The total number of entries in **partition_vector** with values equal to `j` must be equal to the value specified in `partition_sizes[j]`.

The partitioning is typically obtained via some type of mesh partitioner and therefore this information is assumed to be avialable to the calling application, perhaps stored on disk seperately from the global system matrix.

If NULL was passed for this value when the system was read/loaded, trivial partitioning was performed, when block rows are evenly distributed among different ranks (block rows `0..k` go to rank 0, etc)

DESCRIPTION

AMGX_write_system_distributedwrites a linear system of equations, including the matrix, the right hand size, and an optional starting solution vector, from **Matrix** and **Vector** objects to the disk. Unlike **AMGX_write_system**, it takes partitioning information and gather all into a single before printing the full system.

All objects must have previously been created via **AMGX_matrix_create** and **AMGX_vector_create**. In the future, additional file formats may be supported.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

`AMGX_RC_UNKNOWN`

`AMGX_RC_IO_ERROR`

EXAMPLE

```
AMGX_config_handle config;
AMGX_config_create(&config, ""); // use default options
AMGX_resources_handle resources;
int gpu_ids[] = {0};
AMGX_resources_create(&resources, config, MPI_COMM_WORLD, 1, gpu_ids);
AMGX_matrix_handle A;
```

```
AMGX_matrix_create(&A, resources, AMGX_mode_dFFI);
AMGX_vector_handle rhs;
AMGX_vector_handle x;
AMGX_vector_create(&rhs, resources, AMGX_mode_dFFI);
// assume a 4x4 matrix, where rows 0 and 2 belong to partition 0, 1 and 3 to partition 1.
int partition_vector[] = {0,1,0,1};
int partition_sizes[] = {2,2};
AMGX_read_system_distributed(A, rhs, x, argv[1], 1, 2, partition_sizes, 4, partition_vector);
...
AMGX_write_system_distributed(A, rhs, x, "output_system.mtx", 1, 2, partition_sizes, 4, partition_vecto
```

HISTORY

AMGX_write_system_distributed was introduced in API Version 1.

SEE ALSO

Write System, *Read System*, *Read System Distributed*

1.2 Config

NAME

Config

DESCRIPTION

This section describes the API functions for creation and handling of **Config** objects. A **Config** object is a lightweight object that stores a parsed representation of parameter strings. The format allows for nesting and scoped parameters -- see the *Config Syntax* for details.

Config objects are used to store settings for two object types - **Resources** and **Solver** objects. In the case of **Resources**, it specifies information about the communication pattern and other settings that may affect communication between different ranks.

In the case of a **Solver**, the **Config** specifies the algorithm to be employed, as well as all options and parameters to configure that algorithm. Configurations can either be specified via a string or read directly from a text file.

Currently, **Config** objects are not reference counted, so destroying a **Config** object while the it is still being referenced by a **Solver**, **Vector**, **Matrix** or **Resources** will result in undefined behavior. In the future, this behavior may change to generate a run-time error.

AMGX_config_create

AMGX_config_create_from_file

AMGX_config_destroy

HISTORY

Configs were introduced in API Version 1.

SEE ALSO

Confg Create, *Config Create From File*, *Config Destroy*, *Config Syntax*

1.2.1 Config Create

NAME

AMGX_config_create - Creates a **Config** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_config_create(AMGX_config_handle *ret, const char *options);
```

PARAMETERS

ret

Pointer to the opaque handle to be returned. This handle will be valid until it is destroyed via **AMGX_config_destroy**.

options

A string describing the options for this **Config** object. See *Config Syntax* for information on the format.

DESCRIPTION

AMGX_config_create creates a **Config** object, which can then be accessed via the handle returned as **ret**. This version of the creation function takes all of the configuration options as a single null-terminated string. See *Config Syntax* for information on the format.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_CONFIGURATION`

EXAMPLE

```
AMGX_config_handle config;
AMGX_config_create(&config, "algorithm=AGGREGATION,cycle=V");
```

HISTORY

AMGX_config_create was introduced in API Version 1.

SEE ALSO

Config Create From File, *Config Destroy*, *Config Syntax*

1.2.2 Config Create From File

NAME

AMGX_config_create_from_file - Creates a **Config** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_config_create_from_file(AMGX_config_handle *ret,
 const char *param_file);
```

PARAMETERS

ret

Pointer to the opaque handle to be returned. This handle will be valid until it is destroyed via **AMGX_config_destroy**.

options

The file name of a text file to be parsed into this **Config** object.

DESCRIPTION

AMGX_config_create creates a **Config** object, which can then be accessed via the handle returned as **ret**. This version of the creation function takes a file name which specifies a file to be parsed. See *Config Syntax* for information on the format.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_CONFIGURATION`

`AMGX_RC_IO_ERROR`

EXAMPLE

```
AMGX_config_handle config;
AMGX_config_create_from_file(&config, "core/configs/FGMRES");
```

HISTORY

AMGX_config_create was introduced in API Version 1.

SEE ALSO

Config Create, *Config Destroy*, *Config Syntax*

1.2.3 Config Get Default Number Of Rings

NAME

AMGX_config_get_default_number_of_rings - Retrieve the default number of rings.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_config_get_default_number_of_rings(AMGX_config_handle cfg,
 int *num_import_rings);
```

PARAMETERS

cfg

Pointer to the opaque handle that contains the configuration string.

num_import_rings

Default number of rings to be used for a given **Config** object. It is 1 and 2 for the **AMG** solver, when **AGGREGATION** and **CLASSICAL** algorithms are selected, respectively. Also, it is 1 if **AMG** solver is not used.

DESCRIPTION

AMGX_config_get_default_number_of_rings returns the default number of rings to be used for a given **Config** object in the routines such as *Matrix Comm From Maps*, *Matrix Upload All Global*, *Read System Distributed* and *Read System Global*.

This routine only inspects the first two levels of the solver hierarchy. Therefore, it will correctly report number of rings to be used if AMG is used as a solver or a preconditioner, but it will not detect the number of rings correctly for deeper (>2) solver hierarchies. The user is advised to set the number of rings manually in these cases.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_CONFIGURATION`

EXAMPLE

```
AMGX_config_handle config;
int num_import_rings;
AMGX_config_get_default_number_of_rings(config, &num_import_rings);
```

HISTORY

AMGX_config_create was introduced in API Version 2.

SEE ALSO

Matrix Comm From Maps, *Matrix Upload All Global*, *Read System Distributed*, *Read System Global*,

1.2.4 Config Destroy

NAME

AMGX_config_destroy - Destroys a **Config** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_config_destroy(AMGX_config_handle obj);
```

PARAMETERS

obj

Opaque handle specifying the **Config** object to be destroyed.

DESCRIPTION

AMGX_config_destroy destroys an instance of a **Config** object that had been previously created via **AMGX_config_create_from_file** or **AMGX_config_create**. After an instance has been destroyed, subsequent attempts to use the **Config** object will result in undefined behavior.

Note that the **Config** object should not be destroyed until all **Solver**, **Vector**, **Matrix** and **Resources** objects referencing it have been destroyed. Doing so will result in undefinied behavior. Future versions of AMGX may result in a run-time error.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

EXAMPLE

```
AMGX_config_handle config;
AMGX_config_create_from_file(&config, "core/configs/FGMRES");
// use it
AMGX_config_destroy(config);
```

HISTORY

AMGX_config_destroy was introduced in API Version 1.

SEE ALSO

Config Create From File, *Config Create*

1.3 Resources

NAME

Resources

DESCRIPTION

This section describes the API functions for creation and freeing of **Resources** objects. A **Resources** object represents resources that will be used by the local instance of the AMGX library. This includes information about GPUs to use in a multi-GPU system, as well as information about MPI communication in a distributed setting.

The typical lifecycle of a **Resources** will be to create it via **AMGX_resources_create** or **AMGX_resources_create_simple** after the library is initialized. The **Resources** object will be passed to the constructors of all subsequently created **Matrix**, **Vector**, and **Solver** objects to specify where their associated memory will be allocated and how they will communicate with objects in other threads or MPI ranks. At the end of the library usage, the **Resources** object is destroyed via **AMGX_resources_destroy** in the reverse order. That is, it should be destroyed after all **Solver**, **Vector** and **Matrix** objects are destroyed, but before the library is uninitialized.

Any **Solver**, **Vector**, **Matrix**, or objects which interact via **AMGX_vector_bind**, **AMGX_solver_solve**, **AMGX_solver_solve_with_0_initial_guess**, or **AMGX_solver_setup** must all have been bound to the same **Resources** at creation time.

Currently, **Resources** objects are not reference counted, so destroying a **Resources** object while the it is still being referenced by a **Solver**, **Vector** or **Matrix** will result in undefined behavior. In the future, this behavior may change to generate a run-time error.

AMGX_resources_create

AMGX_resources_create_simple

AMGX_resources_destroy

HISTORY

Resources were introduced in API version 2.

SEE ALSO

Resources Create, *Resources Create Simple*, *Resources Destroy*

1.3.1 Resources Create

NAME

AMGX_resources_create - Creates a **Resources** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_resources_create(AMGX_resources_handle *ret,
 AMGX_config_handle resources_config, void *comm, int device_num, const int *devices)
```

PARAMETERS

ret

Pointer to the opaque handle to be returned.

config

The **Config** object which may contain information pertaining to how this **Resources** object will be configured. See below for a desciption of config settings which affect **Resources** creation.

comm

A pointer to the communication specifier. In the case of MPI, this will be a *MPI_Comm **. All ranks in the specified *MPI_Comm* must enter into this routine synchronously since it will perform globally synchronizing operations internally. If this value is NULL (0), it is assumed that only a single process is being used instead, and this routine is being called from a single thread.

device_num

The number of GPU devices which will be utilized by this MPI rank. Currently, only single-GPU per rank is supported, so this value must be 1.

devices

An array of size **device_num** listing the GPU indices which will be used by this rank. For numbering purposes, this corresponds internally to a call to *cudaSetDevice*, so please see the CUDA Programming Guide for information about how physical GPUs are assigned indices.

DESCRIPTION

AMGX_resources_create creates a **Resources** object representing information about the resources used by **Solver**, **Vector**, and **Matrix** objects over their lifetimes. This includes MPI communicators, thread pools, pre-allocated memory buffers, and GPU devices. Certain types of resources may require a license, in which case this routine can fail if the license is insufficient.

The **Resources** object also stores settings that control usage of resources and communication patterns. These settings are passed in via the **confg** input. The settings should all be set in the global scope on the **config** via **AMGX_config_create** or **AMGX_config_create_from_file**. The format of a config string is described in *Config Syntax*. Note that this **Config** may be the same one passed to **AMGX_solver_create**

or it may be separately created - any parameters which are irrelevant to **AMGX_resources_create** will simply be ignored.

When specifying an MPI communicator, all ranks must call this routine synchronously.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_UNKNOWN`

EXAMPLE

For MPI, a **Resources** would be created as follows:

```
int devices[] = {0};
// It is recommended to create a separate communicator to avoid any collisions
MPI_Comm nvamg_comm = ...;
AMGX_resources_handle resource;
AMGX_config_handle config;
AMGX_config_create(config, "communicator=MPI, min_rows_latency_hiding=10000");
AMGX_resources_create(&resource, config, &nvamg_comm, 1, devices);
```

For single-threaded multi-GPU execution, a **Resources** would be created as follows:

```
int devices[] = {0, 1, 2};
AMGX_resources_handle resource;
AMGX_resources_create(&resource, config, NULL, 3, devices);
```

HISTORY

AMGX_resources_create was introduced in API version 2.

SEE ALSO

Resources Create Simple, *Resources Destroy*

1.3.2 Resources Create Simple

NAME

AMGX_resources_create_simple - Creates a **Resources** object in a single-threaded application.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_resources_create_simple(AMGX_resources_handle *ret);
```

PARAMETERS

ret

Pointer to the opaque handle to be returned.

DESCRIPTION

AMGX_resources_create_simple creates a **Resources** object representing information about how data will be stored for subsequently created **Matrix**, **Vector** and **Solver** objects. Unlike **AMGX_resources_create**, this version is suitable for single-GPU and single-threaded applications only.

Calling **AMGX_resources_create_simple** is equivalent to **AMGX_resources_create** with a NULL **comm** parameter and specifying using a single device with id=0.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_UNKNOWN`

EXAMPLE

```
AMGX_resources_handle resources;
AMGX_resources_create_simple(&resources);
```

HISTORY

AMGX_resources_create_simple was introduced in API version 2.

SEE ALSO

AMGX Resources Destroy, *Resources Create*

1.3.3 Resources Destroy

NAME

AMGX_resources_destroy - Destroys a **Resources** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_resources_destroy(AMGX_resources_handle obj);
```

PARAMETERS

obj

Opaque handle specifying the **Resources** object to be destroyed.

DESCRIPTION

AMGX_resources_destroy destroys an instance of a **Resources** object that had been previously created via **AMGX_resources_create** or **AMGX_resources_create_simple**. After an instance has been destroyed, subsequent attempts to use the **Resources** object will result in undefined behavior.

Note that the **Resources** object should not be destroyed until all **Solver**, **Vector** and **Matrix** objects referencing it have been destroyed. Doing so will result in undefinied behavior. Future versions of AMGX may result in a run-time error.

When The **Resources** ojbect has a non-NULL MPI communicator, all ranks must call this routine synchronously.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

EXAMPLE

```
AMGX_resources_handle rsrc;
AMGX_resources_create(&srcs, ...);
// use it
AMGX_resources_destroy(rsrc);
```

HISTORY

AMGX_resources_destroy was introduced in API version 2.

SEE ALSO

Resources Create, *Resources Create Simple*

1.4 Solver

NAME

Solver

DESCRIPTION

This section describes the API functions for creation and handling of **Solver** objects. A **Solver** object is the main object for executing algorithms to solve a linear system of equations. The **Solver** interface is generic, with what type of solution algorithm and what settings to used specified via a **Config** object that is passed to the **Solver** during its initializations. The typical lifecycle of **Solver** will be to create it with a certain configuration via **AMGX_solver_create**, "setup" the solver by passing it the matrix via **AMGX_solver_setup**, and then requesting the solver to compute a solution via either **AMGX_solver_solve** or **AMGX_solver_solve_with_0_initial_guess**.

In the case of multithreaded execution, the **Matrix** and **Vector** objects must have their communication maps initialized prior to being passed to the **Solver**. The **Solver** may initiate communcation and synchronization operations during its setup and solver phases based on the communication maps of the **Matrix** and **Vector** objects. Therefore, all ranks must call **AMGX_solver_setup**, **AMGX_solver_solve**, and **AMGX_solver_solve_with_0_initial_guess** synchronously.

Various information about the solution or the iterations can be queried from the **Solver** after its solve phase completes.

The setup, solve, and query process can be repeated for other matrices or vectors. Alternately, multiple right-hand sides can be processed sequentially by requesting a solution without repeating the setup phase.

Finally, the **Solver** can be destroyed via **AMGX_solver_destroy**. The **Solver** should be destroyed before its bound **Matrix** is destroyed.

A complete discussion of the algorithms which may be employed by a **Solver** may be found in the *Algorithm Guide* section.

AMGX_solver_create

AMGX_solver_destroy

AMGX_solver_setup

AMGX_solver_solve

AMGX_solver_solve_with_0_initial_guess

AMGX_solver_get_iterations_number

AMGX_solver_get_iteration_residual

AMGX_solver_get_status

HISTORY

Solvers were introduced in API Version 1, and modified to support distributed execution in API Version 2.

SEE ALSO

Solver Create, *Solver Destroy*, *Solver Setup*, *Solver Solve*, *Solver Solve With 0 Initial Guess*, *Solver Get Iterations Number*, *Solver Get Iteration Residual*, *Solver Get Status*

1.4.1 Solver Create

NAME

AMGX_solver_create - Creates a **Solver** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_solver_create(AMGX_solver_handle *ret,
  AMGX_resources_handle resources, AMGX_Mode mode,
  const AMGX_config_handle solver_config);
```

PARAMETERS

ret

Pointer to the opaque handle to be returned.

mode

The mode in which the associated **Solver** will operate. The mode value must be consistent with any mode value used to create the **Matrix** or **Vector** objects which will be used to specify the linear system associated with this **Solver**. See below for a desciption of the modes.

resources

The **Resources** object which defines where the memory associated with this object will be allocated, the precision of the associated matrix and vector objects, and any information about how it will communicate with other solvers in other MPI ranks.

config

A handle to a **Config** object that was previously created. Whatever options that are set on the **Config** object at the time it is passed to **AMGX_solver_create** will be used for all subsequent operations by this **Solver**.

DESCRIPTION

AMGX_solver_create creates a solver object to compute the solution to a linear system of equations using the algorithms specified in the **config** object. Note that when it is created, the **Solver** is not bound to any particular **Matrix** object - binding occurs during the case to **AMGX_solver_setup**.

The **mode** parameter can be one of the following values:

```
typedef enum {
  AMGX_mode_hDDI, // 8192
  AMGX_mode_hDFI, // 8448
  AMGX_mode_hFFI, // 8464
  AMGX_mode_dDDI, // 8193
  AMGX_mode_dDFI, // 8449
```

```
   AMGX_mode_dFFI  // 8465
 } AMGX_Mode;
```

For each mode, the first letter h or d specifies whether the matrix data (and subsequent linear solver algorithms) will run on the host or device. The second D or F specifies the precision (double or float) of the **Matrix** data. The third D or F specifies the precision (double or float) of any **Vector** (including right-hand side or unknown vectors). The last I specifies that 32-bit int types are used for all indices. Future versions of AMGX may support additional precisions or mixed precision modes.

All **Matrix** or **Vector** objects attached to this **Solver** must be created with the same **Resources** object.

It is not allowed to call **AMGX_solver_create** multiple times on the same **Solver** without calling **AMGX_solver_destroy** in between. There is no way to change the **Config** on a **Solver** once it is created - instead, you must destroy the **Solver** and create a new one with the modified **Config** settings.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_UNKNOWN`

`AMGX_RC_BAD_MODE`

`AMGX_RC_BAD_PARAMETERS`

EXAMPLE

```
AMGX_config_handle solver_config;
AMGX_config_create_from_file(&solver, "core/configs/V");

AMGX_resources_handle resources;
AMGX_resources_create_simple(&resources);

AMGX_solver_handle solver;
AMGX_solver_create(&solver, resources, AMGX_mode_dDDI, solver_config);
```

HISTORY

AMGX_solver_create was introduced in API Version 1, the calling signature was modified in API Version 2.

SEE ALSO

Solver Destroy, *Resources Create*, *Resources Create Simple*, *Config Create*, *Config Create From File*

1.4.2 Solver Destroy

NAME

AMGX_solver_destroy - Destroys a **Solver** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_solver_destroy(AMGX_solver_handle obj);
```

PARAMETERS

obj

Opaque handle specifying the **Solver** object to be destroyed.

DESCRIPTION

AMGX_solver_destroy destroys an instance of a **Solver** object that had been previously created via **AMGX_solver_create**. After an instance has been destroyed, subsequent attempts to use the **Solver** object will result in undefined behavior.

AMGX_solver_destroy must be called prior to **AMGX_matrix_destroy**.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

EXAMPLE

```
AMGX_solver_handle solver;
AMGX_solver_create(&solver, rsrc, AMGX_mode_dDDI, config);
// use it
AMGX_config_destroy(solver);
```

HISTORY

AMGX_solver_destroy was introduced in API Version 1.

SEE ALSO

Solver Create

1.4.3 Solver Setup

NAME

AMGX_solver_setup - Invoke the setup phase on a **Solver** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_solver_setup(AMGX_solver_handle obj, AMGX_matrix_handle mtx);
```

PARAMETERS

obj

Handle to a **Solver** object that was previously created via **AMGX_solver_create**.

mtx

Handle to a **Matrix** object that was previously created via **AMGX_matrix_create**.

DESCRIPTION

This routine binds a **Matrix** to a **Solver** object and invokes the setup phase of the linear solution algorithm, as defined by the associated **Config** object.

The modes that were used to create **obj** and **mtx** objects must match, otherwise a run-time error will be generated. Similarly, they must all be bound to the same **Resources** object.

Since the setup phase precomputes values that depend on the structure and entries in the matrix, anytime the matrix changes **AMGX_solver_setup** must be called again. At the moment, because the C API does not allow the matrix to be passed in during the solve phase, there is no way to use a setup from a different matrix than is used in the solve phase. In the future, it may be possible to skip repeated calls to **AMGX_solver_setup**, for example if the entries on the matrix change only slightly. If the matrix coefficients were changed via **AMGX_replace_coefficients** but the matrix structure is left unmodified, some algorithms may automatically reuse certain cached computations to result in higher performance. Anytime the communication maps are changed, for example via **AMGX_matrix_comm_from_maps**, **AMGX_solver_setup** must be called again to rebind to the modified **Matrix**.

Repeated calls to **AMGX_solver_setup** are allowed without requiring the **Solver** object to be destroyed and created again. The previously bound **Matrix** will be unbound, and the **Matrix** will be associated with this **Solver** object instead.

In a mulithreaded setting, **AMGX_solver_setup** may perform global synchronization internally. Therefore, all MPI ranks must call this routine synchronously.

A complete description of algorithms employed during setup and solve phases can be found in the *Algorithm Guide* section.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

EXAMPLE

```
AMGX_solver_handle solver;
AMGX_config_handle config;
AMGX_matrix_handle A;
AMGX_vector_handle b, x;
AMGX_resources_handle rsrc;

AMGX_resources_create_simple(rsrc);
AMGX_config_create_from_file(&config, "core/configs/V");
AMGX_solver_create(&solver, rsrc, AMGX_mode_dDDI, config);
AMGX_matrix_create(&A, rsrc, AMGX_mode_dDDI);
AMGX_vector_create(&b, rsrc, AMGX_mode_dDDI);
AMGX_vector_create(&x, rsrc, AMGX_mode_dDDI);
AMGX_read_system(A, b, NULL, "test.mtx");
AMGX_solver_setup(solver, A);
AMGX_solver_solve(solver, b, x);
```

HISTORY

AMGX_solver_setup was introduced in API Version 1.

SEE ALSO

Solver Solve, *Solver Solve With 0 Initial Guess*, *Algorithm Guide*.

1.4.4 Solver Solve With 0 Initial Guess

NAME

AMGX_solver_solve_with_0_initial_guess - Invoke the solve phase on a **Solver** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_solver_solve_with_0_initial_guess(AMGX_solver_handle obj,
 AMGX_vector_handle rhs, AMGX_vector_handle sol);
```

PARAMETERS

obj

Handle to a **Solver** object that was previously created via **AMGX_solver_create** and bound to a **Matrix** via **AMGX_solver_setup**.

rhs

Handle to a **Vector** object that was previously created via **AMGX_vector_create**. The vector represents the right-hand side of the equation to be solved. In an MPI or multi-GPU setting, **rhs** must have communication maps or partition information already set via **AMGX_vector_bind** or **AMGX_read_system_distributed**.

sol

Handle to a **Vector** object that was previously created via **AMGX_vector_create**. The vector represents the solution vector to the equation to be solved. Note that the value of **sol** will be ignored, as it is assumed to be the zero vector. This presents a slight opportunity for internal optimizations since some of the calculations can be simplified in this case. In an MPI or multi-GPU setting, **sol** must have communication maps or partition information already set via **AMGX_vector_bind** or **AMGX_read_system_distributed**.

DESCRIPTION

This routine invokes the solve phase of a linear system solution. Different from **AMGX_solver_solve**, it ignores the values in the specified **sol Vector** and assumes that the start point for the iterations will be the zero-vector. This may be more efficient in some circumstances.

The modes that were used to create **obj**, **rhs**, and **sol** objects must match, otherwise a run-time error will be generated. Similarly, the **Resources** instance to which they are bound must all match.

Attempts to call **AMGX_solver_solve_with_0_initial_guess** without previously calling **AMGX_solver_setup** will result in a run-time error. The solve phase may be invoked multiple times for different **rhs** and **sol** values without calling **AMGX_solver_setup** in between.

AMGX_solver_solve_with_0_initial_guess will run the designated iterative solver until the stopping criteria is met. Which iterative solver and stopping criteria to use are defined in the **Config** object that was passed to **AMGX_solver_create**. See the *Algorithm Guide* for a discussion of the different algorithms that may be used and their settings. The solve phase will terminate when one of the following conditions are met:

1. The number of iterations exceeds **max_iters**.

2. The residual is below the convergence criteria. What criteria and comparison to use is determined by **convergence** and the value that is considered converged is determined by **tolerance**. What norm to use is determined by the **norm** setting.

The iterative solver will ignore the value of **sol**, as it is assumed to be the zero vector.

Note that the matrix is not required to be passed in because it was previously set via **NAMG_solver_setup**. However, AMGX currently does not employ reference counting. Therefore, calls to **AMGX_solver_solve_with_0_initial_guess** will result in undefined behavior if the referenced **Matrix** has been destroyed after being passed to the solver via **AMGX_solver_setup**. In the future, this behavior may change.

In an MPI setting, **AMGX_solver_solve_with_0_initial_guess** may perform global synchronization internally. Therefore, all MPI ranks must call this routine synchronously.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

EXAMPLE

```
// A minimal example of reading in a matrix and rhs and solving it.
AMGX_solver_handle solver;
AMGX_config_handle config;
AMGX_matrix_handle A;
AMGX_vector_handle x,b;
AMGX_resources_handle rsrc;


AMGX_resources_create_simple(rsrc);
AMGX_config_create_from_file(&config, "core/configs/V");
AMGX_solver_create(&solver, rsrc, AMGX_mode_dDDI, config);
AMGX_matrix_create(&A, rsrc, AMGX_mode_dDDI);
AMGX_vector_create(&x, rsrc, AMGX_mode_dDDI); // x created but not initialized
AMGX_vector_create(&b, rsrc, AMGX_mode_dDDI);
AMGX_read_system(A, b, NULL, "test.mtx"); // will not initialize x either
AMGX_solver_setup(solver, A);
AMGX_solver_solve_with_0_initial_guess(solver, b, x);
```

HISTORY

AMGX_solver_solve was introduced in API Version 1.

SEE ALSO

Solver Setup, *Solver Solve*, *Algorithm Guide*

1.4.5 Solver Solve

NAME

AMGX_solver_solve - Invoke the solve phase on a **Solver** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_solver_solve(AMGX_solver_handle obj, AMGX_vector_handle rhs,
  AMGX_vector_handle sol);
```

PARAMETERS

obj

Handle to a **Solver** object that was previously created via **AMGX_solver_create** and bound to a **Matrix** via **AMGX_solver_setup**.

rhs

Handle to a **Vector** object that was previously created via **AMGX_vector_create**. The vector represents the right-hand side of the equation to be solved. In an MPI or multi-GPU setting, **rhs** must have communication maps or partition information already set via **AMGX_vector_bind** or **AMGX_read_system_distributed**.

sol

Handle to a **Vector** object that was previously created via **AMGX_vector_create**. The vector represents the solution vector to the equation to be solved. Note that the value of **sol** will be used as the starting point for the iterative algorith, so its value should be initialized via **AMGX_vector_upload**, **AMGX_vector_set_zero**, **AMGX_read_system**, or **AMGX_read_system_distributed**. In an MPI or multi-GPU setting, **sol** must have communication maps or partition information already set via **AMGX_vector_bind** or **AMGX_read_system_distributed**.

DESCRIPTION

This routine invokes the solve phase of a linear system solution. Different from **AMGX_solver_solve_with_0_initial_guess**, it uses the values in the specified **sol Vector** as a starting point for the iterative procedure. This allows for intellgent stopping and restarting of iterative solutions. However, it may be less efficient on a per-iteration basis than **AMGX_solver_solve_with_0_initial_guess**.

The modes that were used to create **obj**, **rhs**, and **sol** objects must match, otherwise a run-time error will be generated. Similarly, the **Resources** instance to which they are bound must all match.

Attempts to call **AMGX_solver_solve** without previously calling **AMGX_solver_setup** will result in a run-time error. The solve phase may be invoked multiple times for different **rhs** and **sol** values without calling **AMGX_solver_setup** in between.

AMGX_solver_solve will run the designated iterative solver until the stopping criteria is met. Which iterative solver and stopping criteria to use are defined in the **Config** object that was passed to **AMGX_**

solver_create. See the *Algorithm Guide* for a discussion of the different algorithms that may be used and their settings. The solve phase will terminate when one of the following conditions are met:

1. The number of iterations exceeds **max_iters**.

2. The residual is below the convergence criteria. What criteria and comparison to use is determined by **convergence** and the value that is considered converged is determined by **tolerance**. What norm to use is determined by the **norm** setting.

The iterative solver will not initialize the value of **sol**, so whatever value is passed in will be used as the starting point for the iterative algorithm.

Note that the matrix is not required to be passed in because it was previously set via **NAMG_solver_setup**. However, AMGX currently does not employ reference counting. Therefore, calls to **AMGX_solver_solve** will result in undefined behavior if the referenced **Matrix** has been destroyed after being passed to the solver via **AMGX_solver_setup**. In the future, this behavior may change.

In an MPI setting, **AMGX_solver_solve** may perform global synchronization internally. Therefore, all MPI ranks must call this routine synchronously.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

EXAMPLE

```
// A minimal example of reading in a matrix and rhs and solving it.
AMGX_solver_handle solver;
AMGX_config_handle config;
AMGX_matrix_handle A;
AMGX_vector_handle x,b;
AMGX_resources_handle rsrc;


AMGX_resources_create_simple(rsrc);
AMGX_config_create_from_file(&config, "core/configs/V");
AMGX_solver_create(&solver, rsrc, AMGX_mode_dDDI, config);
AMGX_matrix_create(&A, rsrc, AMGX_mode_dDDI);
AMGX_vector_create(&x, rsrc, AMGX_mode_dDDI);
AMGX_vector_create(&b, rsrc, AMGX_mode_dDDI);
AMGX_read_system(A, b, NULL, "test.mtx");
AMGX_solver_setup(solver, A);
int n, block_dim;
AMGX_matrix_get_size(A, &n, &block_dim, &block_dim);
AMGX_vector_set_zero(x, n, block_dim);
AMGX_solver_solve(solver, b, x);
```

HISTORY

AMGX_solver_solve was introduced in API Version 1.

SEE ALSO

Solver Setup, *Solver Solve With 0 Initial Guess*, *Algorithm Guide*

1.4.6 Solver Get Iterations Number

NAME

AMGX_solver_get_iterations_number - Return the number of iterations that were executed during the last solve phase.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_solver_get_iterations_number(AMGX_solver_handle obj, int *n);
```

PARAMETERS

obj

Handle to a **Solver** object that was previously created via **AMGX_solver_create**.

n

Pointer to an int which will be set to the returned value.

DESCRIPTION

This returns the number of iterations which were used to reach a stopping point (either achieving convergence or reaching the maximum number of allowed iterations) during the most recent call to **AMGX_solver_solve** or **AMGX_solver_solve_with_0_initial_guess**. Calling this routine before either of those routines results in `*n` being set to 0.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

EXAMPLE

```
AMGX_solver_setup(solver, A);
AMGX_solver_solve_with_0_initial_guess(solver, b, x);
int n;
AMGX_solver_get_iterations_number(solver, &n);
```

HISTORY

AMGX_solver_get_iterations_number was introduced in API Version 1.

SEE ALSO

Solver Solve, *Solver Solve With 0 Initial Guess*, *Solver Get Iteration Residual*, *Solver Get Status*

1.4.7 Solver Get Iteration Residual

NAME

AMGX_solver_get_iteration_residual - Return the value of the residual for a given iteration from the last solve phase.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_solver_get_iteration_residual(AMGX_solver_handle obj, int iter,
 int idx, double *res);
```

PARAMETERS

obj

Handle to a **Solver** object that was previously created via **AMGX_solver_create**.

iter

The value of the iteration's residual to inspect. If this value is out of range, it will result in a run-time error.

idx

The index of the entry in the block of the residual to retrieve. For example, for a 4x4 block system, the residual will have 4 components, and this value can be in the range from 0 to 3. For a scalar sytem, this value should be 0. If this value is out of range, a run-time error will be returned.

res

A pointer to the double variable which will be set to the requested residual value.

DESCRIPTION

This returns the value of a residual from a specific iteration during the most recent call to **AMGX_solver_solve** or **AMGX_solver_solve_with_0_initial_guess**.

Calling this routine with out of range **idx** or **iter** values results in an error and `*res` will be set to -1. Calling it before a solve phase has executed means that the **iter** value is automatically out of range.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

EXAMPLE

```
AMGX_solver_setup(solver, A);
AMGX_solver_solve_with_0_initial_guess(solver, b, x);
int n;
int mtx_size, block_dim;
AMGX_matrix_get_size(A, &mtx_size, &block_dim, &block_dim);
AMGX_solver_get_iterations_number(solver, &n);
for (int idx=0; idx < block_dim; idx++) {
  double final_residual;
  AMGX_solver_get_iteration_residual(solver, n, idx, &final_residual);
  printf("Final residual, component %d: %f\n", idx, final_residual);
}
```

HISTORY

AMGX_solver_get_iteration_residual was introduced in API Version 1.

SEE ALSO

Solver Solve, *Solver Solve With 0 Initial Guess*, *Solver Get Iterations Number*, *Solver Get Status*

1.4.8 Solver Get Status

NAME

AMGX_solver_get_status - Retrieve the status flag from the last solve phase.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_solver_get_status(AMGX_solver_handle solver,
 AMGX_SOLVE_STATUS* status);
```

PARAMETERS

obj

Handle to a **Solver** object that was previously created via **AMGX_solver_create**.

status

Pointer to a variable which will be set with the **Solver** object's status.

DESCRIPTION

Retrieve the status flag specifying the result of the previous call to **AMGX_solver_solve** or **AMGX_solver_solve_with_0_initial_guess**.

The flag will be one of

```
enum AMGX_SOLVE_STATUS {
  AMGX_SOLVE_SUCCESS=0,
  AMGX_SOLVE_FAILED=1,
  AMGX_SOLVE_DIVERGED=2,
}
```

`AMGX_SOLVE_SUCCESS` means the solver achieved convergence with no errors.

`AMGX_SOLVE_FAILED` means the solver failed for some reason.

`AMGX_SOLVE_DIVERGED` means that the solver didn't fail, but it reached the maximum iteration count before the residual dropped low enough to be considered converged.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

EXAMPLE

```
AMGX_solver_setup(solver, A);
AMGX_solver_solve(solver, b, x);
AMGX_SOLVE_STATUS status;
AMGX_solver_get_status(solver, &status);
```

HISTORY

AMGX_solver_get_status was introduced in API Version 1.

SEE ALSO

Solver Solve, *Solver Solve With 0 Initial Guess*, *Solver Get Iterations Number*, *Solver Get Iteration Residual*

1.5 Matrix

NAME

Matrix

DESCRIPTION

This section describes the API functions for creation and handling of **Matrix** objects. A **Matrix** object represents a sparse linear system of equations that is stored on either the host or the device.

The **Matrix** object is primarily a way to manage data that is passed from the application into the AMGX library. It is assumed that the calling application is responsible for creating the matrix data via application-specific logic - AMGX is not designed to assist in this aspect of numerical computation, as there are no routines for manipulating matrices or executing BLAS or other mathematical routines on them.

To aid in the easy integration of AMGX into existing applications, AMGX has flexible support for a variety of layouts and matrix types, including support for block systems and different schemes for storing the diagonals. It also allows for matrices to be distributed across multiple MPI ranks (distributed memory) or partitioned across multiple GPUs attached to the same MPI rank. There is support specifying connectivity between matrix partitions on different MPI ranks in a variety of ways.

In a single-threaded applicaton, the typical lifecycle of **Matrix** will be to create it via **AMGX_matrix_create**, associating it with a **Resources** instance that specifies the GPU to be utilized by this thread. Matrix buffers are then transferred from the application into AMGX via **AMGX_matrix_upload_all**, or read from disk via **AMGX_read_system**. Once the data has been loaded, the coefficients may be updated without affecting the non-zero structure of the matrix via **AMGX_matrix_replace_coefficients**. The **Matrix** will be bound to a **Solver** object via **AMGX_solver_setup**, and the associated linear system will be solved via **AMGX_solver_solve** or **AMGX_solver_solve_with_0_initial_guess**. Finally, the memory associated with this matrix may be deallocated after the solver completes via **AMGX_matrix_destroy**.

In an MPI application, the lifecycle will be to create it via **AMGX_matrix_create**, associating it with a **Resources** instance that specifies the the GPU to be utilized and the MPI communicator through which it can find ranks containing other portions of the matrix. Before data can be uploaded to the **Matrix**, the application must first **AMGX_matrix_comm_from_maps** or **AMGX_matrix_comm_from_maps_one_ring**. Data is then uploaded via **AMGX_matrix_upload_all**. Alternately, the matrix structure can be read from disk via **AMGX_read_system_distributed**. The coefficients may be changed without affecting the non-zero structure via **AMGX_matrix_replace_coefficients**. If the non-zero structure does change, then the caller must specify the new communication maps via **AMGX_matrix_comm_from_maps_one_ring** or **AMGX_matrix_comm_from_maps** before calling **AMGX_matrix_replace_coefficients**. The **Matrix** will be passed to a **Solver** object via **AMGX_solver_setup**. Finally, the memory may be deallocated after the **Solver** is destroyed via **AMGX_matrix_destroy**.

Currently, the **Matrix** objects are not reference counted, so destroying a **Matrix** object while the it is still being used by a **Solver** will result in undefined behavior. In the future, this behavior may change to generate a run-time error.

AMGX_matrix_create

AMGX_matrix_destroy

AMGX_matrix_upload_all

AMGX_matrix_replace_coefficients

AMGX_matrix_get_size

AMGX_matrix_comm_from_maps

AMGX_matrix_comm_from_maps_one_ring

HISTORY

Matrices were introduced in API version 1. Support for distributed and multi-GPU applications was added in API version 2.

SEE ALSO

Matrix Create, *Matrix Destroy*, *Matrix Upload All*, *Matrix Replace Coefficients*, *Matrix Get_size*, *Matrix Comm From Maps*, *Matrix Comm From Maps One Ring*, *Matrix Sort*

1.5.1 Matrix Create

NAME

AMGX_matrix_create - Creates a **Matrix** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_matrix_create(AMGX_matrix_handle *ret,
    AMGX_resources_handle resources, AMGX_Mode mode);
```

PARAMETERS

ret

Pointer to the opaque handle to be returned.

resources

The **Resources** object which defines where the memory associated with this object will be allocated and information about how it will communicate with other matrices in other MPI ranks or threads.

mode

The mode in which the associated **Matrix** will operate. The mode value must be consistent with any mode value used to create the **Solver** or **Vector** objects which will be used to solve the linear system associated with this **Matrix**. See below for a desciption of the modes.

DESCRIPTION

AMGX_matrix_create creates a matrix object to handle matrix data.

The **mode** parameter can be one of the following values:

```
typedef enum {
  AMGX_mode_hDDI, // 8192
  AMGX_mode_hDFI, // 8448
  AMGX_mode_hFFI, // 8464
  AMGX_mode_dDDI, // 8193
  AMGX_mode_dDFI, // 8449
  AMGX_mode_dFFI  // 8465
} AMGX_Mode;
```

For each mode, the first letter h or d specifies whether the matrix data (and subsequent linear solver algorithms) will run on the host or device. The second D or F specifies the precision (double or float) of the **Matrix** data. The third D or F specifies the precision (double or float) of any **Vector** (including right-hand side or unknown vectors). The last I specifies that 32-bit int types are used for all indices. Future versions of AMGX may support additional precisions or mixed precision modes.

Note that AMGX does not currently perform any automatic precision conversion, so the data that is passed into a **Matrix** object via subsequent calls to **AMGX_matrix_upload_all** or **AMGX_matrix_replace_coefficients** must match the precision (specified in the second position) of the **mode** parameter when the **Matrix** object was created.

RETURN VALUES

Relevant return values:

AMGX_RC_OK

AMGX_RC_UNKNOWN

AMGX_RC_BAD_MODE

AMGX_RC_BAD_PARAMETERS

EXAMPLE

```
AMGX_resources_handle resources;
AMGX_matrix_handle matrix;
AMGX_resources_create_simple(&resources);
AMGX_matrix_create(&matrix, resources, AMGX_mode_dDFI);
```

HISTORY

AMGX_matrix_create was introduced in API Version 1, the calling signature was modified in API Version 2.

SEE ALSO

Matrix Destroy, *Resources Create*, *Resources Create Simple*

1.5.2 Matrix Destroy

NAME

AMGX_matrix_destroy - Destroys a **Matrix** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_matrix_destroy(AMGX_matrix_handle obj);
```

PARAMETERS

obj

Opaque handle specifying the **Matrix** object to be destroyed.

DESCRIPTION

AMGX_matrix_destroy destroys an instance of a **Matrix** object that had been previously created via **AMGX_matrix_create**. After an instance has been destroyed, subsequent attempts to use the **Matrix** object will result in undefined behavior. Currently, AMGX does not employ reference counting. Therefore, calls to **AMGX_solver_solve** or **AMGX_solver_solve_with_0_initial_guess** will result in undefine behavior after the referenced **Matrix** has been destroyed. In the future, this behavior may change.

AMGX_matrix_destroy must be called after **AMGX_solver_destroy**.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

EXAMPLE

```
AMGX_resources_handle resources;
AMGX_matrix_handle matrix;
AMGX_resources_create_simple(&resources);
AMGX_matrix_create(&matrix, resources, AMGX_mode_dDFI);

// use it

AMGX_matrix_destroy(matrix);
```

HISTORY

AMGX_matrix_destroy was introduced in API Version 1.

SEE ALSO

Matrix Create

1.5.3 Matrix Upload All

NAME

AMGX_matrix_upload_all - Copy data (packed) local matrix into a **Matrix** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_matrix_upload_all(AMGX_matrix_handle mtx, int n,
 int nnz, int block_dimx, int block_dimy, const int *row_ptrs,
 const int *col_indices, const void *data, const void *diag_data);
```

PARAMETERS

mtx

Opaque handle specifying the **Matrix** object.

n

The dimension of the matrix in terms of block-units. For single-threaded operation, AMGX only allows square matrices, so this is both the number of columns and rows. In MPI operation, a matrix may have more columns that rows to reference halo vertices on other partitions. In this case **n** represents the number of block-rows, and any column indices referencing a column greater than or equal to `n` will refer to a halo reference.

nnz

The number of non-zero entries in the CSR matrix, in terms of block units. For example, a 4x4 matrix with 10 columns/rows, with 3 non-zero blocks per row, would have nnz = 30, even though there are a total of 480 entries (16 per block, 30 total blocks).

block_dimx

The blocksize in x direction. For a scalar matrix, this value should be 1. Currently only square blocks are supported, so **block_dimx** and **block_dimy** must be equal.

block_dimy

The blocksize in y direction. For a scalar matrix, this value should be 1. Currently only square blocks are supported, so **block_dimx** and **block_dimy** must be equal.

row_ptrs

Array of indices into the col_indices structure. **row_ptrs** has `n+1` entries. Entry `i` indicates the starting index of the values belonging to row `i` in the **col_indices** table. `row_ptrs[0]` must always be 0, and `row_ptrs[n]` must always be equal to **nnz**.

col_indices

Array of the column index of the nonzero blocks in the matrix. **col_indices** must have **nnz** entries. `col_indices[i]` contains the column index (in block units) of nonzero block `i`.

data

Array of the matrix entries in "array of structures" (AoS) layout. **data** must have `nnz * block_dimx * block_dimy` entries, where **block_dimx** and **block_dimy** are, for example, both 4 in the case of a 4x4 block matrix, or 1 in the case of a scalar matrix. `data[i*block_dimx*block_dimy]` contains the entry 0,0 in block `i`. `data[i*block_dimx*block_dimy+1*block_dimy]` contains the entry 1,0 in block `i`, and so on. Data within the block is assumed to be arranged in row-major scanline order.

diag_data

Optional array of external diagonal entries for each row. If there is no external diagonal, in other words, the diagonal is contained in the matrix itself the diag_data must be set to a null pointer. If this value is non-null, it is assumed to be an array with `n*block_dimx*block_dimy` entries in AoS layout. `diag_data[i*block_dimx*block_dimy]` is the 0,0 entry in the i,i block in the matrix.

DESCRIPTION

AMGX_matrix_upload_all copies data (packed) local matrix to a **Matrix** object from the application. When this routine is called, the buffers will be allocated to the required size (if necessary), and the data will be copied into the **Matrix** data structure.

In single node setting, no packing is needed and the local matrix is the entire coefficient matrix of the linear system at hand.

In distributed setting, the packed local matrix is obtained from the global coefficient matrix on rank `i` following these steps:

1. Select the rows (with global column indices) belonging to rank `i` and read them into memory

2. Pack the columns:

a) Reorder the column indices such that the columns corresponding to the global diagonal elements present on rank `i` come first.

b) Then, reorder the remaining column indices, in the order to which they belong (have connections) to rank's `i` neighbors. Columns belonging to neighbor 0 first, neighbor 1 second, etc., leaving the natural ordering of columns within the same neighbor.

c) Notice that we now have a rectangular matrix with reordered column indices that are numbered locally. To keep track of this reordering create a local_to_global map which indicates how the locally renumbered columns map into their global counterparts.

3. Reorder the rows and columns (such that the rows with no connections to neighbors come first):

a) Find the reordering that moves rows with no connections to neighbors first

b) Apply it to rows and columns (belonging to the global diagonal elements present on rank `i` only, in other words, columns up to **n**).

4. Pass the resulting packed local matrix to this routine.

The user buffers may reside on the host or device. The library will internally take advantage of Unified Virtual Addressing (UVA). This feature is available starting with CUDA Toolkit 4.0 release, on 64-bit Linux and Windows (TCC) platforms, with compute capability 2.0 and higher Tesla class GPUs. These minimum settings are required for the library to work correctly.

If the user buffers are on the host and the **Matrix** mode indicates device storage (first letter is a d), the

copy will transfer data to the GPU.

It is recommended that the host buffers passed to **AMGX_matrix_upload_all** be pinned previously via **AMGX_pin_memory**. This allows the underlying CUDA driver to achieve higher data transfer rates across the PCI-Express bus. This routine and the underlying memory transfers will run synchronously. In other words, when the call to **AMGX_matrix_upload_all** returns, the copy is guaranteed to have been completed. Future versions of AMGX may add functionality to allow for asynchronous copies.

The precision of all floating point buffers must match the **mode** parameter that was set when this **Matrix** object was created.

It is legal to call **AMGX_matrix_upload_all** on a matrix more than once, even if the **n** and **nnz** values have changed. The buffers will be resized appropriately.

In the MPI distributed setting, **AMGX_matrix_comm_from_maps_one_ring** or **AMGX_matrix_comm_from_maps** must be called before calling this routine. Calls to any of these routines will only take effect after the next call to **AMGX_matrix_upload_all**.

It is valid to change whether a **Matrix** has non-NULL **diag_data** between one call and the next. That is, a **Matrix** may have **diag_data** on one call to **AMGX_matrix_upload_all** and have none on a subsequent call, or vice-versa. Note that AMGX does not check if diagonal entries are included in the regular row data as well as being listed separately. So if **diag_data** is non-NULL, it is up to the application to ensure that these entries do not also appear in the **row_ptrs**, **col_indices**, and **data** buffers. Repeating the diagonal entries will likely result in undefined behavior.

If the **Matrix** is bound to a **Resources** object with a non-NULL MPI communicator and has communication maps, all ranks must call this routine synchronously because it may perform communication or synchronization internally.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

`AMGX_RC_NO_MEMORY`

`AMGX_RC_UNKNOWN`

EXAMPLE (single node)

```
// define the 2x2 matrix with 2x2 blocks:
// (1 -2)   0
// (-3 1)
//        (1 -4)
//   0    (-5 1)


float data[]  = {1, -2, -3, 1, 1, -4, -5, 1};
int col_ind[] = {0, 1};
```

```
int row_ptr[] = {0, 1, 2};

AMGX_matrix_handle matrix;
AMGX_resources_handle rsrc;
AMGX_resources_create_simple(&rsrc);
AMGX_matrix_create(&matrix, rsrc, AMGX_mode_dFFI);
AMGX_matrix_upload_all(matrix, 2, 2, 2, 2, row_ptr, col_ind, data, 0);
```

EXAMPLE (distributed setting)

```
// define the global 4x4 matrix with 1x1 blocks:
// (1 -2) (   1)
// (-3 1) (    )
// (    ) (1 -4)
// (   1) (-5 1)

// consider 2 MPI ranks
if (rank == 0) {
  //packed local matrix, based on permutation = [1, 0] and local_to_global = [0, 1, 3]
  // (1 -3) (0)
  // (-2 1) (1)
  float data[]  = {1, -3, -2, 1, 1};
  int col_ind[] = {0,  1,  0, 1, 2};
  int row_ptr[] = {0, 2, 5};
}
if (rank == 1) {
  //packed local matrix, based on permutation = [0, 1] and local_to_global = [2 3 1]
  // (1 -4) (0)
  // (-5 1) (1)
  float data[]  = {1, -4, -5, 1, 1};
  int col_ind[] = {0,  1,  0, 1, 2};
  int row_ptr[] = {0, 2, 5};
}
AMGX_matrix_handle matrix;
AMGX_resources_handle rsrc;
AMGX_resources_create_simple(&rsrc);
AMGX_matrix_create(&matrix, rsrc, AMGX_mode_dFFI);
AMGX_matrix_upload_all(matrix, 2, 5, 1, 1, row_ptr, col_ind, data, 0);
```

HISTORY

AMGX_matrix_upload_all was introduced in API Version 1.

SEE ALSO

Matrix Create, *Matrix Replace Coefficients*, *Pin Memory*, *Unpin Memory*, *Matrix Comm From Maps*, *Matrix Comm From Maps One Ring*, *Read System Maps One Ring*

1.5.4 Matrix Upload All Global

NAME

AMGX_matrix_upload_all_global - Copy data (with global column indices) into a **Matrix** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_matrix_upload_all_global(AMGX_matrix_handle mtx,
 int n_global, int n, int nnz, int block_dimx, int block_dimy,
 const int *row_ptrs, const void *col_indices_global, const void *data, const void *diag_data,
 int allocated_halo_depth, int num_import_rings, const int *partition_vector);
```

PARAMETERS

mtx

Opaque handle specifying the **Matrix** object.

n_global

The dimension of the global matrix in terms of block-units. This also corresponds to the number of columns (in block-units) in this partition.

n

The dimension of the local matrix in terms of block-units. This also corresponds to the number of rows (in block-units) in this partition.

nnz

The number of non-zero entries in the local CSR matrix, in terms of block-units.

block_dimx

The blocksize in x direction. For a scalar matrix, this value should be 1. Currently only square blocks are supported, so **block_dimx** and **block_dimy** must be equal. Currently only **block_dimx**=1 is supported.

block_dimy

The blocksize in y direction. For a scalar matrix, this value should be 1. Currently only square blocks are supported, so **block_dimx** and **block_dimy** must be equal. Currently only **block_dimy**=1 is supported.

row_ptrs

Array of indices into the **col_indices_global** structure. **row_ptrs** has `n+1` entries. Entry `i` indicates the starting index of the values belonging to row `i` in the **col_indices_global** table. `row_ptrs[0]` must always be 0, and `row_ptrs[n]` must always be equal to **nnz**.

col_indices_global

Array of the `global` column indices of the nonzero blocks in the matrix. The type of column indices must be 64-bit integer (int64_t). **col_indices_global** must have **nnz** entries. `col_indices_global[i]` contains the column index (in block-units) of nonzero block `i`.

data

Array of the matrix entries in "array of structures" (AoS) layout. **data** must have `nnz * block_dimx * block_dimy` entries, where **block_dimx** and **block_dimy** are, for example, both 4 in the case of a 4x4 block matrix, or 1 in the case of a scalar matrix. `data[i*block_dimx*block_dimy]` contains the entry 0,0 in block `i`. `data[i*block_dimx*block_dimy+1*block_dimy]` contains the entry 1,0 in block `i`, and so on. Data within the block is assumed to be arranged in row-major scanline order.

diag_data

Optional array of external diagonal entries for each row. If there is no external diagonal, in other words, the diagonal is contained in the matrix itself the diag_data must be set to a null pointer. If this value is non-null, it is assumed to be an array with `n*block_dimx*block_dimy` entries in AoS layout. `diag_data[i*block_dimx*block_dimy]` is the 0,0 entry in the i,i block in the matrix. Currently external diagonal is not supported.

allocated_halo_depth

In order to support halo exchanges with overlap, it is necessary to allocate buffers to store halo vertices and connectivity data. The **allocated_halo_depth** setting causes the **Matrix** to allocate enough storage to support halo exchanges for halos up to the specified depth. The actual amount of overlap is specified via a setting in the **Resources** and **Solver** object. Currently, **allocated_halo_depth** must be equal to **num_import_rings**. In the future, AMGX may support computing extra layers of halo regions automatically.

num_import_rings

The number of rings of overlap to be specified. The first ring will have depth of 1, the second will have depth of 2, and so on. This allows the caller to provide the library with information about overlap regions with depth greater than 1. In this case, "depth" refers to the number of edges that must be traversed on the matrix connectivity graph in order to reach a non-halo vertex. In other words, the first ring (depth=1) contains halo vertices, where each vertex is directly connected to at least one non-halo vertex. The second ring contains halo vertices where each vertex is directly connected to at least one of the vertices in the first ring, and so on.

partition_vector

An array of partition assignments of the global matrix. The array must have size equal to **n_global**. Each entry `partition_vector[i]` will be an integer between 0 and `num_partitions-1` indicating the partition to which block-row `i` belongs. The partitioning is typically obtained via some type of mesh partitioner and therefore this information is assumed to be avialable to the calling application, perhaps stored on disk seperately from the global system matrix.

If NULL is passed for this value, trivial partitioning is performed, when block-rows are evenly distributed among different ranks (block rows `0..k` go to rank 0, etc)

DESCRIPTION

AMGX_matrix_upload_all_global copies data (with global column indices) to a **Matrix** object from the application. When this routine is called, the buffers will be allocated to the required size (if necessary), and the data will be copied into the **Matrix** data structure.

The user buffers may reside on the host or device. The library will internally take advantage of Unified Virtual Addressing (UVA). This feature is available starting with CUDA Toolkit 4.0 release, on 64-bit Linux and Windows (TCC) platforms, with compute capability 2.0 and higher Tesla class GPUs. These minimum settings are required for the library to work correctly.

If the user buffers are on the host and the **Matrix** mode indicates device storage (first letter is a d), the copy will transfer data to the GPU.

It is recommended that the host buffers passed to **AMGX_matrix_upload_all** be pinned previously via **AMGX_pin_memory**. This allows the underlying CUDA driver to achieve higher data transfer rates across the PCI-Express bus. This routine and the underlying memory transfers will run synchronously. In other words, when the call to **AMGX_matrix_upload_all** returns, the copy is guaranteed to have been completed. Future versions of AMGX may add functionality to allow for asynchronous copies.

The precision of all floating point buffers must match the **mode** parameter that was set when this **Matrix** object was created.

It is legal to call **AMGX_matrix_upload_all_global** on a matrix more than once, even if the **n** and **nnz** values have changed. The buffers will be resized appropriately.

There is no need to call **AMGX_matrix_comm_from_maps_one_ring** or **AMGX_matrix_comm_from_maps** before calling this routine. These calls will be done implicitly inside.

It is valid to change whether a **Matrix** has non-NULL **diag_data** between one call and the next. That is, a **Matrix** may have **diag_data** on one call to **AMGX_matrix_upload_all_global** and have none on a subsequent call, or vice-versa. Note that AMGX does not check if diagonal entries are included in the regular row data as well as being listed separately. So if **diag_data** is non-NULL, it is up to the application to ensure that these entries do not also appear in the **row_ptrs**, **col_indices_global**, and **data** buffers. Repeating the diagonal entries will likely result in undefined behavior.

If the **Matrix** is bound to a **Resources** object with a non-NULL MPI communicator and has communication maps, all ranks must call this routine synchronously because it may perform communication or synchronization internally.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

`AMGX_RC_NO_MEMORY`

`AMGX_RC_UNKNOWN`

EXAMPLE (distributed setting)

```
// define the global 4x4 matrix with 1x1 blocks:
// (1 -2) (   1)
// (-3 1) (    )
// (    ) (1 -4)
// (   1) (-5 1)


// consider 2 MPI ranks
if (rank == 0) {
```

```
  // (1 -2) (   1)
  // (-3 1) (    )
  float data[]         = {1, -2, 1, -3, 1};
  int col_ind_global[] = {0,  1,  3, 0, 1};
  int row_ptr[]        = {0, 3, 5};
}
if (rank == 1) {
  // (    ) (1 -4)
  // (   1) (-5 1)
  float data[]         = {1, -4, 1, -5, 1};
  int col_ind_global[] = {2,  3,  1, 2, 3};
  int row_ptr[]        = {0, 2, 5};
}
AMGX_matrix_handle matrix;
AMGX_resources_handle rsrc;
AMGX_resources_create_simple(&rsrc);
AMGX_matrix_create(&matrix, rsrc, AMGX_mode_dFFI);
AMGX_matrix_upload_all_global(matrix, 4, 2, 5, 1, 1, row_ptr, col_indices_global,
                              data, NULL, 1 /*or 2*/, 1 /*or 2*/, NULL);
```

HISTORY

AMGX_matrix_upload_all_global was introduced in API Version 2.

SEE ALSO

Matrix Create, *Matrix Replace Coefficients*, *Pin Memory*, *Unpin Memory*, *Matrix Comm From Maps*, *Matrix Comm From Maps One Ring*

1.5.5 Matrix Replace Coefficients

NAME

AMGX_matrix_replace_coefficients - Copy coefficient data into a **Matrix** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_matrix_replace_coefficients(AMGX_matrix_handle mtx, int n,
 int nnz, const void *data, const void *diag_data);
```

PARAMETERS

mtx

Opaque handle specifying the **Matrix** object.

n

The dimension of the matrix in terms of block-units. AMGX only allows square matrices, so this is both the number of columns and rows. This value must match the previous dimension passed in via **AMGX_matrix_upload_all**.

nnz

The number of non-zero entries in the CSR matrix, in terms of block units. For example, a 4x4 matrix with 10 columns/rows, with 3 non-zero blocks per row, would have nnz = 30, even though there are a total of 480 entries (16 per block, 30 total blocks). This value must match the previous dimension passed in via **AMGX_matrix_upload_all**.

data

Array of the matrix entries in "array of structures" (AoS) layout. **data** must have `nnz * block_dimx * block_dimy` entries, where **block_dimx** and **block_dimy** are, for example, both 4 in the case of a 4x4 block matrix, or 1 in the case of a scalar matrix. `data[i*block_dimx*block_dimy]` contains the entry 0,0 in block `i`. `data[i*block_dimx*block_dimy+1*block_dimy]` contains the entry 1,0 in block `i`, and so on. Data within the block is assumed to be arranged in row-major scanline order.

diag_data

Optional array of external diagonal entries for each row. If there is no external diagonal, in other words, the diagonal is contained in the matrix itself the diag_data must be set to a null pointer. If this value is non-null, it is assumed to be an array with `n*block_dimx*block_dimy` entries in AoS layout. `diag_data[i*block_dimx*block_dimy]` is the 0,0 entry in the i,i block in the matrix.

DESCRIPTION

Uploads new coefficient data to a **Matrix** without changing its non-zero structure. Since **AMGX_matrix_replace_coefficients** does not allocate buffers, **AMGX_matrix_upload_all** must have been called previously on the **Matrix** object. Attempts to call **AMGX_matrix_replace_coefficients** on a **Matrix** where **AMGX_matrix_upload_all** had previously been invoked will result in an error if the values of **n** and **nnz** do not match those previously.

The user buffers may reside on the host or device. The library will internally take advantage of Unified Virtual Addressing (UVA). This feature is available starting with CUDA Toolkit 4.0 release, on 64-bit Linux and Windows (TCC) platforms, with compute capability 2.0 and higher Tesla class GPUs. These minimum settings are required for the library to work correctly.

If the user buffers are on the host and the **Matrix** mode indicates device storage (first letter is a d), the copy will transfer data to the GPU.

The meaning and layout of the parameters is the same as **AMGX_matrix_upload_all**. As with **AMGX_matrix_upload_all**, **diag_data** is an optional argument.

RETURN VALUES

Relevant return values:

AMGX_RC_OK

AMGX_RC_BAD_PARAMETERS

AMGX_RC_BAD_MODE

AMGX_RC_UNKNOWN

EXAMPLE

```
// define the 2x2 matrix with 2x2 blocks:
// (1 -2)   0
// (-3 1)
//        (1 -4)
//   0    (-5 1)


float data[] = {1, -2, -3, 1, 1, -4, -5, 1};
int col_ind[] = {0, 1};
int row_ptr[] = {0, 1, 2};


AMGX_matrix_handle matrix;
AMGX_resources_handle rsrc;
AMGX_resources_create_simple(&rsrc);
AMGX_matrix_create(&matrix, rsrc, AMGX_mode_dFFI);
AMGX_matrix_upload_all(matrix, 2, 2, 2, 2, row_ptr, col_ind, data, 0);


// replace with an identity matrix
float data2 = {1, 0, 0, 1, 1, 0, 0, 1};
AMGX_matrix_replace_coefficients(matrix, 2, 2, data2, 0);
```

HISTORY

AMGX_matrix_replace_coefficients was introduced in API Version 1.

SEE ALSO

Matrix Upload All, *Pin Memory*, *Unpin Memory*

1.5.6 Matrix Get Size

NAME

AMGX_matrix_get_size - Get size information from a **Matrix**.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_matrix_get_size(const AMGX_matrix_handle obj, int *n,
 int *block_dimx, int *block_dimy);
```

PARAMETERS

mtx

Opaque handle specifying the **Matrix** object.

n

Pointer to an int which will be set to the matrix dimension in block-units. AMGX only allows square matrices, so this is both the number of columns and rows. In multithreaded execution, where column indices may index entries larger than the number of rows, this will return the number of rows in the local matrix partition.

block_dimx

Pointer to an int which will be set to the size in x of a matrix block.

block_dimy

Pointer to an int which will be set to the size in y of a matrix block.

DESCRIPTION

AMGX_matrix_get_size retrieves the dimensions of a matrix in terms of block units and the size of each block. This is useful when the matrix is loaded from disk, and the application needs to create vector of the same size.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

EXAMPLE

```
AMGX_matrix_handle matrix;
AMGX_matrix_create(&matrix, AMGX_mode_dFFI);

// ...fill the matrix with values

int n, blockx, blocky;
AMGX_matrix_get_size(matrix, &n, &blockx, &blocky);
```

HISTORY

AMGX_matrix_get_size was introduced in API Version 1.

SEE ALSO

Matrix Create

1.5.7 Matrix Comm From Maps

NAME

AMGX_matrix_comm_from_maps - Given a **Matrix** object corresponding to a local partition, creates the communication maps necessary for distributed operation, allowing for arbitrary amounts of overlap.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_matrix_comm_from_maps(AMGX_matrix_handle mtx_part,
 int allocated_halo_depth, int num_import_rings,
 int max_num_neighbors, const int *neighbors,
 const int *send_ptrs, int const *send_maps,
 const int *recv_ptrs, int const *recv_maps);
```

PARAMETERS

mtx_part

The **Matrix** corresponding to the local partition of a distributed matrix. This routine assumes that the matrix has already been partitioned across multiple MPI ranks, and that this matrix contains only rows which are local to this MPI rank. The matrix is assumed to be rectangular, with more columns than rows. If the number of block-rows is M and the number of block-columns is N, with M < N, then indices M through N-1 refer to "halo" vertices. Halo vertices represent a local copy of data (matrix row or vector entries) which are located on a remote MPI rank.

allocated_halo_depth

In order to support halo exchanges with overlap, it is necessary to allocate buffers to store halo vertices and connectivity data. The **allocated_halo_depth** setting causes the **Matrix** to allocate enough storage to support halo exchanges for halos up to the specified depth. The actual amount of overlap is specified via a setting in the **Resources** and **Solver** object. Currently, **allocated_halo_depth** must be equal to **num_import_rings**. In the future, AMGX may support computing extra layers of halo regions automatically.

num_import_rings

The number of rings of overlap to be specified. The first ring will have depth of 1, the second will have depth of 2, and so on. This allows the caller to provide the library with information about overlap regions with depth greater than 1. In this case, "depth" refers to the number of edges that must be traversed on the matrix connectivity graph in order to reach a non-halo vertex. In other words, the first ring (depth=1) contains halo vertices, where each vertex is directly connected to at least one non-halo vertex. The second ring contains halo vertices where each vertex is directly connected to at least one of the vertices in the first ring, and so on.

max_num_neighbors

The number of different neighbors which can be referenced by a single ring. Since different rings may reference indices which exist on different numbers of neighboring MPI ranks, this should be the maximum value. This can happen, say, if there is a narrow region on one of the neighboring matrix partions, and the wavefront of increasing halo rings actually passes through the neighbor into a neighbor of that neighbor.

neighbors

An array of size **max_num_neighbors** listing the index of each MPI rank that is referenced by a halo vertex in one of the rings on this rank. In other words, an ordered list of MPI ranks which will exchange data via halo exchanges.

send_ptrs

An array of size `max_num_neighbors * num_import_rings +1`. The format is similar to the "row pointers" data in a typical CSR matrix. The value of entry `i*max_num_neighbors+ j` will be the position in the **send_maps** array where data corresponding to ring `i`, neighbor `neighbors[j]` begins. If that data set is empty, for example because a particular ring has no connections with a particular neighbor, the index should be equal to the index of the next stored region, as with an empty row in a CSR matrix. The first entry, `send_ptrs[0]`, should always be 0. The last entry, `send_ptrs[max_num_neighbors*num_import_rings]`, should be the number of total entries in the **send_maps** array.

send_maps

An array of size `send_ptrs[max_num_neighbors*num_import_rings]` containing data corresponding to ring and neighbor entries. The entries beginning at index `send_ptrs[i*max_num_neighbors+j]` and ending at index `send_ptrs[i*max_num_neighbors+j+1]-1` (inclusive) correspond to ring `i` and neighbor `neighbors[j]`. This sequence of entries are interpreted as a "map" specifying the local row indices from this matrix partition which will be sent to the MPI rank `neighbors[j]`. The data corresponding to these local row indices will be packed into a transfer buffer, and then sent to the corresponding MPI rank. The order in which the local row indices are listed corresponds to the order in which they will be packed into the transfer buffer.

recv_ptrs

An array of size `max_num_neighbors * num_import_rings +1`. The format is identical to the **send_ptrs** array, except referring to the **recv_maps** array which lists the transfer buffers which are to be received from each neighboring partition. The last entry, `recv_ptrs[max_num_neighbors*num_import_rings]`, should be the number of total entries in the **recv_maps** array.

recv_maps

An array of size `recv_ptrs[max_num_neighbors*num_import_rings]` containing data corresponding to ring and neighbor entries. The layout of this array is identical to the **send_maps** array, except that the beginning and ending entries of each region are specified by the **recv_ptrs** array. The entries beginning at index `recv_ptrs[i*max_num_neighbors+j]` and ending at index `recv_ptrs[i*max_num_neighbors+j+1]-1` (inclusive) correspond to ring `i` and neighbor `neighbors[j]`. This sequence of entries is interpreted as a "map" specifying the local halo indices from this matrix partition which will be received from the MPI rank *neighbors[j]*. The data received from `neighbor[i]` will have been packed into a transfer buffer in the order specified by that remote matrix partition's **send_maps** value corresponding to this (local) MPI rank. The order in which the indices appear in **send_maps** must therefore correspond to this order.

DESCRIPTION

AMGX_matrix_comm_from_maps defines communication connections between a local matrix partition and one or more remote matrix partitions. Unlike **AMGX_matrix_comm_from_maps_one_ring**, this routine allows the caller to specify arbitrary numbers of halo rings, assuming they are known by the application. While AMGX is able to discover these extra halo rings itself, if the application already knows the halo

connections the construction of internal communication maps can be optimized. Therefore **AMGX_matrix_comm_from_maps_one_ring** and **AMGX_matrix_comm_from_maps** result in equivalent functionality, but **AMGX_matrix_comm_from_maps** may be more efficient.

Because the **AMGX_matrix_comm_from_maps** routine may apply reordering or other optimizations based on communication patterns, this routine must be called *before* **AMGX_matrix_upload_all**. Calling it *after* **AMGX_matrix_upload_all** will result in the **Matrix** to not actually have any attached communication information. In other words, a call to **AMGX_matrix_comm_from_maps** only records the communication information, but this information isn't actually uploaded and applied to the **Matrix** until a subsequent call to **AMGX_matrix_upload_all**. Because the communication maps depend on the nonzero structure of the matrix, calling **AMGX_matrix_upload_all** again without calling **AMGX_matrix_comm_from_maps_one_ring** or **AMGX_matrix_comm_from_maps** will invalidate the communication maps and will also result in a no communication. Changes to the coefficients via **AMGX_matrix_replace_coefficients** are allowed after calling **AMGX_matrix_comm_from_maps** and will have no effect on the communication maps.

If **AMGX_matrix_comm_from_maps** is called on a **Matrix**, those communication maps must be transfered to any right-hand side or solution **Vector** objects associated with the **Matrix** by calling **AMGX_vector_bind**.

In an MPI setting, communications between matrices must be created via **AMGX_matrix_comm_from_maps** or **AMGX_matrix_comm_from_maps_one_ring** before **AMGX_solver_setup** is called on a **Matrix**.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_UNKNOWN`

`AMGX_RC_BAD_MODE`

`AMGX_RC_BAD_PARAMETERS`

EXAMPLE

```
// Create matrices with overlap of depth 2,
// corresponding to a Laplacian operator on the following grid:
// 0 - 1 - 2 - 3
// {0,1} are on rank 0, and {2,3} are on rank 1.
// on rank 0, ring 0 = {2}, ring 1 = {3}
// on rank 1, ring 0 = {1}, ring 1 = {0}
AMGX_resources_handle rsrc;
AMGX_resources_create(&rsrc, ...);
AMGX_matrix_handle matrix;
AMGX_matrix_create(&matrix, rsrc, AMGX_mode_dFFI);


if (MPI_rank == 0) {
  float data[] = {2, -1, -1, 2, -1, -1, 2, -1};
```

```
  float cols[] = {0,  1,  0, 1,  2,  1, 2,  3};
  //                                ^^^^^^^^- this is a halo row (not local)
  float rows[] = {0,2,5,8};

  int nbrs[] = {1};
  int send_ptrs[] = {0,1,2}; // ring 0 spans pos [0,1), ring 1 spans pos [1,2)
  int send_maps[] = {1,0};   // ring 0: local idx 1 (=> global idx 1)
                             // ring 1: local idx 0 (=> global idx 0)
  int recv_ptrs[] = {0,1,2}; // ring 0 spans pos [0,1), ring 1 spans pos [1,2)
  int recv_maps[] = {2,3};   // ring 0: local idx 2 (=> global idx 2)
                             // ring 1: local idx 3 (=> global idx 3)
  AMGX_matrix_comm_from_maps(matrix, 1, 2, 2, 1, nbrs,
       send_ptrs, send_maps, recv_sizes, recv_maps);
  AMGX_matrix_upload_all(matrix, 3, 8, 1, 1, rows, cols, data, 0);
}
else if (MPI_rank == 1) {
  float data[] = {-1, 2, -1, -1, 2, -1, 2, -1};
  float cols[] = {2,  0,  1, 0,  1,  1, 2,  3};
  //                                ^^^^^^^^ halo row
  float rows[] = {0,3,5,8};

  int nbrs[] = {1};
  int send_ptrs[] = {0,1,2}; // ring 0 spans pos [0,1), ring 1 spans pos [1,2)
  int send_maps[] = {0,1};   // ring 0: local idx 0 (=> global idx 2)
                             // ring 1: local idx 1 (=> global idx 3)
  int recv_ptrs[] = {0,1,2}; // ring 0 spans pos [0,1), ring 1 spans pos [1,2)
  int send_maps[] = {2,1};   // ring 0: local idx 2 (=> global idx 1)
                             // ring 1: local idx 3 (=> global idx 0)
  AMGX_matrix_comm_from_maps(matrix, 1, 2, 2, 1, nbrs,
       send_ptrs, send_maps, recv_sizes, recv_maps);
  AMGX_matrix_upload_all(matrix, 3, 8, 1, 1, rows, cols, data, 0);
}
```

HISTORY

AMGX_matrix_comm_from_maps was introduced in API version 2.

SEE ALSO

Matrix Comm From Maps One Ring, *Matrix Upload All*, *Read System Distributed*, *Vector Bind*

1.5.8 Matrix Comm From Maps One Ring

NAME

AMGX_matrix_comm_from_maps_one_ring - Given a **Matrix** object corresponding to a local partition, creates the communication maps necessary for distributed operation, with only one layer of overlap.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_matrix_comm_from_maps_one_ring(AMGX_matrix_handle mtx_part,
 int allocated_halo_depth,
 int num_neighbors, const int *neighbors,
 const int *send_sizes, int const **send_maps,
 const int *recv_sizes, int const **recv_maps);
```

PARAMETERS

mtx_part

The **Matrix** corresponding to the local partition of a distributed matrix. This routine assumes that the matrix has already been partitioned across multiple MPI ranks, and that this matrix contains only rows which are local to this MPI rank. The matrix is assumed to be rectangular, with more columns than rows. If the number of block-rows is N and the number of block-columns is M, with M > N, indices N through M-1 refer to "halo" vertices. Halo vertices represent a local copy of data (matrix row or vector entries) which are located on a remote MPI rank.

allocated_halo_depth

In order to support halo exchanges with overlap, it is necessary to allocate buffers to store halo vertices and connectivity data. The **allocated_halo_depth** setting causes the **Matrix** to allocate enough storage to support halo exchanges for halos up to the specified depth. The actual amount of overlap is specified via a setting in the **Resources** and **Solver** object. Currently, **allocated_halo_depth** must be equal to `num_import_rings` and therefore here it equals 1 (which is implicit from the name of the routine). In the future, AMGX may support computing extra layers of halo regions automatically.

num_neighbors

The number of MPI ranks which share a boundary with this rank. In other words, the number of MPI ranks which will exchange data via halo exchanges.

neighbors

An array of size **num_neighbors** listing the index of each MPI rank that shares a boundary with this rank. In other words, a list of MPI ranks which will exchange data via halo exchanges.

send_sizes

An array of size **num_neighbors**. The value in entry `i` is the number of local (i.e. non-halo) rows in this rank's matrix partition which will be sent to the MPI rank `neighbors[i]`.

send_maps

An array of size **num_neighbors** of arrays, where entry `i` is another array of size `send_sizes[i]`. Array `i` is a "map" specifying the local row indices from this matrix partition which will be sent to

the MPI rank `neighbors[i]`. The data corresponding to these local row indices will be packed into a transfer buffer, and then sent to the corresponding MPI rank. The order in which the local row indices are listed corresponds to the order in which they will be packed into the transfer buffer. For simple cases with `n` local rows, this will be an offset mapping, where local index `n+j` (which is a halo index) will map to position `j` in the transfer buffer.

recv_sizes

An array of size **num_neighbors**. The value in entry `i` is the number of non-local (i.e. halo) rows in this rank's matrix partition which will be received from the MPI rank `neighbors[i]`.

recv_maps

An arry of size **num_neighbors** of arrays, where entry `i` is another array of size `recv_sizes[i]`. Array `i` is a "map" specifying the local halo indices from this matrix partition which will be received from the MPI rank `neighbors[i]`. The data received from `neighbor[i]` will have been packed into a transfer buffer in the order specified by that remote matrix partition's **send_maps** value corresponding to this (local) MPI rank. The order in which the indices appear in **send_maps** must therefore correspond to this order.

DESCRIPTION

AMGX_matrix_comm_from_maps_one_ring defines communication connections between a local matrix partition and one or more remote matrix partitions, in the typical case where there is only a single level of overlap between the matrix connectivity graphs. In other words, when interpreting the matrices as graphs, all halo vertices are directly connected to at least one non-halo (local) vertex.

Because the **AMGX_matrix_comm_from_maps_one_ring** routine may apply reordering or other optimizations based on communication patterns, this routine must be called *before* **AMGX_matrix_upload_all**. Calling it *after* **AMGX_matrix_upload_all** will result in the **Matrix** to not actually have any attached communication information. In other words, a call to **AMGX_matrix_comm_from_maps_one_ring** only records the communication information, but this information isn't actually uploaded and applied to the **Matrix** until a subsequent call to **AMGX_matrix_upload_all**.

Because the communication maps depend on the non-zero structure of the matrix, calling **AMGX_matrix_upload_all** again without calling **AMGX_matrix_comm_from_maps_one_ring** or **AMGX_matrix_comm_from_maps** will invalidate the communication maps and will also result in a no communication. Changes to the coefficients via **AMGX_matrix_replace_coefficients** are allowed after calling **AMGX_matrix_comm_from_maps** and will have no effect on the communication maps.

If **AMGX_matrix_comm_from_maps_one_ring** is called on a **Matrix**, those communication maps must be transfered to any right-hand side or solution **Vector** objects associated with the **Matrix** by calling **AMGX_vector_bind**.

In an MPI setting, communications between matrices must be created via **AMGX_matrix_comm_from_maps**, or **AMGX_matrix_comm_from_maps_one_ring** before **AMGX_solver_setup** is called on a **Matrix**.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

AMGX_RC_UNKNOWN

AMGX_RC_BAD_MODE

AMGX_RC_BAD_PARAMETERS

EXAMPLE

```
// Create matrices corresponding to a Laplacian operator on the following grid:
// 0 - 1 - 2 - 3
// {0,1} are on rank 0, and {2,3} are on rank 1.
// node 2 is a halo node for rank 0, and node 1 is a halo node for rank 1.
AMGX_resources_handle rsrc;
AMGX_resources_create(&rsrc, ...);
AMGX_matrix_handle matrix;
AMGX_matrix_create(&matrix, rsrc, AMGX_mode_dFFI);

if (MPI_rank == 0) {
  float data[] = {2, -1, -1, 2, -1};
  float cols[] = {0,  1,  0, 1,  2}; // last entry refers to a halo point (col idx 2)
  float rows[] = {0,2,5};

  int nbrs[] = {1};
  int send_sizes[] = {1};
  int send_map[] = {1}; // local idx 1 (=> global idx 1) gets packed to the xfer buffer
  int recv_sizes[] = {1};
  int recv_map[] = {2}; // halo idx 2 (=> global idx 2) comes from remote xfer buffer
  AMGX_matrix_comm_from_maps_one_ring(
    matrix, 1, nbrs, send_sizes, &send_map, recv_sizes, &recv_map);
  AMGX_matrix_upload_all(matrix, 2, 5, 1, 1, rows, cols, data, 0);
}
else if (MPI_rank == 1) {
  float data[] = {-1, 2, -1, -1, 2};
  float cols[] = {2,  0,  1, 0,  1}; // first entry refers to a halo point (col idx 2)
  float rows[] = {0,3,5};

  int nbrs[] = {1};
  int send_sizes[] = {1};
  int send_map[] = {0}; // local idx 0 (=> global idx 2) gets packed to xfer buffer
  int recv_sizes[] = {1};
  int recv_map[] = {2}; // halo idx 2 (=> global idx 1) comes from remote xfer buffer
  AMGX_matrix_comm_from_maps_one_ring(
    matrix, 1, nbrs, send_sizes, &send_map, recv_sizes, &recv_map);
  AMGX_matrix_upload_all(matrix, 2, 5, 1, 1, rows, cols, data, 0);
}
```

HISTORY

AMGX_matrix_comm_from_maps_one_ring was introduced in API version 2.

SEE ALSO

Matrix Comm From Maps, *Matrix Upload All*, *Read System Distributed*, *Vector Bind*

1.6 Vector

NAME

Vector

DESCRIPTION

This section describes the API functions for creation and handling of **Vector** objects. A **Vector** object represents a linear algebra vector that is stored on either the host or the device. In AMGX accessed via the C API, vectors are mostly opaque objects. Via the lower-level C++ API, it is possible to manipulate vectors and execute BLAS-like routines on them. However, via the C-API, the **Vector** object is primarily a way to manage data that is passed from the application into the AMGX library and other utility operations are not supported.

In a single-threaded application, the typical lifecycle of **Vector** will be to create it via **AMGX_vector_create**, associating it with a **Resources** instance that specifies the GPUs to be utilized by this thread. Note that the **Matrix** and **Vector** must be associated with the same **Resources** object. The vector buffer is then transferred from the application into AMGX via **AMGX_vector_upload_all**, setting it to zeroes via **AMGX_vector_set_zero**, or reading from disk via **AMGX_read_system**. The **Vector** will be passed to an **Solver** object via **AMGX_solver_solve** or **AMGX_solver_solve_with_0_initial_guess**. For a **Vector** representing the solution to the linear system, its content can be retrieved from AMGX via **AMGX_vector_download**. Finally, the memory associated with this vector may be deallocated after the solver completes via **AMGX_vector_destroy**.

In an MPI application, it is necessary to associate communication data with the **Vector** object. A **Vector** will be created via **AMGX_vector_create**, associating it with a **Resources** instance that specifies the GPUs to be utilized and the MPI communicator across which other partitions of the vector will be distributed. The application establishes communication information on a **Matrix** via **AMGX_matrix_comm_from_maps** or **AMGX_matrix_comm_from_maps_one_ring**. The communication information is then propagated to any vectors which will be associated with this particular linear system (as a right-hand side or solution) via **AMGX_vector_bind**. Note that the **Matrix** and **Vector** objects must be associated with the same **Resources** object for this to be valid. Then, as in the single-threaded case, the **Vector** will be passed to an **Solver** object via **AMGX_solver_solve** or **AMGX_solver_solve_with_0_initial_guess**. For a **Vector** representing the solution to the linear system, its content can be retrieved from AMGX via **AMGX_vector_download**. Finally, the memory associated with this vector may be deallocated after the solver completes via **AMGX_vector_destroy**.

Currently, the **Vector** objects are not reference counted, so destroying a **Vector** object while the it is still being used by a **Solver** will result in undefined behavior. In the future, this behavior may change to generate a run-time error.

AMGX_vector_create

AMGX_vector_destroy

AMGX_vector_upload

AMGX_vector_set_zero

AMGX_vector_download

AMGX_vector_get_size

AMGX_vector_bind

HISTORY

Vectors were introduced in API version 1. Support for distributed applications was added in API version 2.

SEE ALSO

Vector Create, Vector Destroy, Vector Upload, Vector Set Zero, Vector Download, Vector Get Size, Vector Bind

1.6.1 Vector Create

NAME

AMGX_vector_create - Creates a **Vector** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_vector_create(AMGX_vector_handle *ret,
    AMGX_resources_handle resources, AMGX_Mode mode);
```

PARAMETERS

ret

Pointer to the opaque handle to be returned.

resources

The **Resources** object which defines where the memory associated with this object will be allocated, the precision of this vector object, and any information about how it will communicate with other vectors in other MPI ranks or threads.

mode

The mode in which the associated **Vector** will operate. The mode value must be consistent with any mode value used to create the **Solver** or **Matrix** objects which will be used to solve the linear system associated with this **Vector**. See below for a desciption of the modes.

DESCRIPTION

AMGX_vector_create creates a matrix object to handle vector data.

The **mode** parameter can be one of the following values:

```
typedef enum {
  AMGX_mode_hDDI, // 8192
  AMGX_mode_hDFI, // 8448
  AMGX_mode_hFFI, // 8464
  AMGX_mode_dDDI, // 8193
  AMGX_mode_dDFI, // 8449
  AMGX_mode_dFFI  // 8465
} AMGX_Mode;
```

For each mode, the first letter h or d specifies whether the matrix data (and subsequent linear solver algorithms) will run on the host or device. The second D or F specifies the precision (double or float) of the **Matrix** data. The third D or F specifies the precision (double or float) of any **Vector** (including right-hand side or unknown vectors). The last I specifies that 32-bit int types are used for all indices. Future versions of AMGX may support additional precisions or mixed precision modes.

Note that AMGX does not currently perform any automatic precision conversion, so the data that is passed into a **Vector** object via subsequent calls to **AMGX_vector_upload** and **AMGX_vector_download** must match the precision (specified in the third position) of the **mode** parameter when the **Vector** was created.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_UNKNOWN`

`AMGX_RC_BAD_MODE`

EXAMPLE

```
AMGX_resources_handle resources;
AMGX_resources_create_simple(&resources);

AMGX_vector_handle vector;
AMGX_vector_create(&vector, resources, AMGX_mode_dDFI);
```

HISTORY

AMGX_vector_create was introduced in API Version 1, the calling signature was modified in API Version 2.

SEE ALSO

Vector Destroy, *Resources Create*, *Resources Create Simple*

1.6.2 Vector Destroy

NAME

AMGX_vector_destroy - Destroys a **Vector** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_vector_destroy(AMGX_vector_handle obj);
```

PARAMETERS

obj

Opaque handle specifying the **Vector** object to be destroyed.

DESCRIPTION

AMGX_vector_destroy destroys an instance of a **Vector** object that had been previously created via **AMGX_vector_create**. After an instance has been destroyed, subsequent attempts to use the **Vector** object will result in undefined behavior. Currently, AMGX does not employ reference counting. Therefore, calls to **AMGX_solver_solve** or **AMGX_solver_solve_with_0_initial_guess** will result in undefine behavior after the referenced right-hand side or solution **Matrix** objects have been destroyed. In the future, this behavior may change.

It is recommended **AMGX_vector_destroy** to be called after **AMGX_solver_destroy** and prior to **AMGX_matrix_destroy**.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

EXAMPLE

```
AMGX_vector_handle vector;
AMGX_vector_create(&vector, AMGX_mode_dDDI);
// use it
AMGX_vector_destroy(vector);
```

HISTORY

AMGX_vector_destroy was introduced in API Version 1.

SEE ALSO

Vector Create

1.6.3 Vector Upload

NAME

AMGX_vector_upload - Copy data into a **Vector** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_vector_upload(AMGX_vector_handle vec, int n, int block_dim,
 const void *data);
```

PARAMETERS

vec

Opaque handle specifying the **Vector** object.

n

The number of entries to be copied into the **Vector** object, in block units.

block_dim

The number of values per block.

data

Pointer to the beginning of the user array to be copied. The array must have `n*block_dim` entries, organize in "array of structure" format. In other words, for a 4x4 block system, **block_dim** would be 4, and the value for entry `j` within block `i` would be at position `i*4+j`.

DESCRIPTION

AMGX_vector_upload copies data from a user buffer into a **Vector** object. When this routine is called, the buffer will be allocated to the required size (if necessary), and the data will be copied into the **Vector** data structure.

The user buffers may reside on the host or device. The library will internally take advantage of Unified Virtual Addressing (UVA). This feature is available starting with CUDA Toolkit 4.0 release, on 64-bit Linux and Windows (TCC) platforms, with compute capability 2.0 and higher Tesla class GPUs. These minimum set tings are required for the library to work correctly.

If the user buffers are on the host and the **Vector** mode indicates device storage (first letter is a d), the copy will transfer data to the GPU.

It is recommended that the host buffers passed to **AMGX_vector_upload** be pinned previously via **AMGX_pin_memory**. This allows the underlying CUDA driver to achieve higher data transfer rates across the PCI-Express bus. This routine and the underlying memory transfers will run synchronously. In other words, when the call to **AMGX_vector_upload** returns, the copy is guaranteed to have been completed. Future versions of AMGX may add functionality to allow for asynchronous copies.

The precision of all the floating point buffer must match the **mode** parameter that was set when this **Vector** object was created.

It is legal to call **AMGX_vector_set_zero** or **AMGX_vector_upload** on a vector more than once, even if the **n** and **block_dim** values have changed. The buffers will be resized appropriately.

In an MPI or multi-GPU settting, the **Vector** must have communication maps and partition information set on it before calling **AMGX_vector_upload** by first calling **AMGX_vector_bind**. The maps will be applied during the uploading process. Calling **AMGX_vector_bind** after **AMGX_vector_upload** will result in the communication maps and partition information not being applied until the next time **AMGX_vector_upload** or **AMGX_vector_set_zero** is called.

RETURN VALUES

Relevant return values:

AMGX_RC_OK

AMGX_RC_BAD_PARAMETERS

AMGX_RC_BAD_MODE

AMGX_RC_NO_MEMORY

AMGX_RC_UNKNOWN

EXAMPLE

```
AMGX_vector_handle vector;
AMGX_vector_create(&vector, resources, AMGX_mode_dDFI);
float data[] = {1, 2, 3, 4, 1, 2, 3, 4};
AMGX_pin_memory(data);
// set the block vector  with 2 entries, block size 4: [(1 2 3 4) (1 2 3 4)]
AMGX_vector_upload(vector, 2, 4, data);
```

HISTORY

AMGX_vector_upload was introduced in API Version 1.

SEE ALSO

Vector Create, *Vector Download*, *Vector Set Zero*, *Pin Memory*, *Unpin Memory*, *Vector Bind*, *Matrix Comm From Maps*, *Matrix Comm From Maps One Ring*

1.6.4 Vector Download

NAME

AMGX_vector_download - Retrieve data from a **Vector** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_vector_download(const AMGX_vector_handle vec, void *data);
```

PARAMETERS

vec

Opaque handle specifying the **Vector** object.

data

Pointer to the beginning of the user array to receive the data. The array must have enough space to store `n*block_dim` entries, where **n** and **block_dim** can be retrieved via **AMGX_vector_get_size**.

DESCRIPTION

AMGX_vector_download copies data a **Vector** object into a user buffer.

The user buffers may reside on the host or device. The library will internally take advantage of Unified Virtual Addressing (UVA). This feature is available starting with CUDA Toolkit 4.0 release, on 64-bit Linux and Windows (TCC) platforms, with compute capability 2.0 and higher Tesla class GPUs. These minimum settings are required for the library to work correctly.

If the user buffers are on the host and the **Vector** mode indicates device storage (first letter is a d), the copy will transfer data from the GPU.

It is recommended that the host buffers passed to **AMGX_vector_upload** be pinned previously via **AMGX_pin_memory**. This allows the underlying CUDA driver to achieve higher data transfer rates across the PCI-Express bus. This routine and the underlying memory transfers will run synchronously. In other words, when the call to **AMGX_vector_download** returns, the copy is guaranteed to have been completed. Future versions of AMGX may add functionality to allow for asynchronous copies.

The precision of the data in the floating point buffer will match the **mode** parameter that was set when this **Vector** object was created.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

AMGX_RC_BAD_MODE

AMGX_RC_UNKNOWN

EXAMPLE

```
AMGX_vector_handle vector;
AMGX_vector_create(&vector, resources, AMGX_mode_dDDI);
AMGX_vector_set_zero(vector, 100, 1);
double data[100];
AMGX_pin_memory(data);
AMGX_vector_download(vector, data);
```

HISTORY

AMGX_vector_download was introduced in API Version 1.

SEE ALSO

Vector Upload, *Vector Set Zero*, *Pin Memory*, *Unpin Memory*

1.6.5 Vector Set Zero

NAME

AMGX_vector_set_zero - Allocate storage if needed and set all of the values in a **Vector** to zero.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_vector_set_zero(AMGX_vector_handle vec, int n, int block_dim);
```

PARAMETERS

vec

Opaque handle specifying the **Vector** object to be updated.

n

The number of entries in the **Vector** object, in block units.

block_dim

The number of values per block.

DESCRIPTION

AMGX_vector_set_zero sets the data in **Vector** object to all zeros. When this routine is called, the buffer will be allocated to the required size (if necessary). The semantics of **AMGX_vector_set_zero** are identical to calling <AMGX_vector_upload> with a buffer of the same size consisting of all zeroes.

The precision of the floating point buffer will match the **mode** parameter that was set when this **Vector** object was created.

It is legal to call **AMGX_vector_set_zero** or **AMGX_vector_upload** on a matrix more than once, even if the **n** and **block_dim** values have changed. The buffers will be resized appropriately.

In an MPI or multi-GPU settting, the **Vector** must have communication maps and partition information set on it before calling **AMGX_set_zero** by first calling **AMGX_vector_bind**. The maps will be applied when the data is set. Calling **AMGX_vector_bind** after **AMGX_vector_upload** will result in the communication maps and partition information not being applied until the next time **AMGX_vector_upload** or **AMGX_vector_set_zero** is called.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

`AMGX_RC_BAD_MODE`

AMGX_RC_NO_MEMORY

AMGX_RC_UNKNOWN

EXAMPLE

```
AMGX_vector_handle vector;
AMGX_vector_create(&vector, AMGX_mode_dDFI);
// set the block vector  with 2 entries, block size 4: [(0 0 0 0) (0 0 0 0)]
AMGX_vector_set_zero(vector, 2, 4);
```

HISTORY

AMGX_vector_set_zero was introduced in API Version 1.

SEE ALSO

Vector Create, *Vector Upload*, *Vector Comm From Matrix*, *Vector Bind*, *Matrix Comm From Maps*, *Matrix Comm From Maps One Ring*

1.6.6 Vector Get Size

NAME

AMGX_vector_size - Retrieve size from a **Vector** object.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_vector_get_size(const AMGX_vector_handle vec, int *n,
 int *block_dim);
```

PARAMETERS

vec

Opaque handle specifying the **Vector** object.

n

Pointer to the variable which will be set to the size of this *Vector* in block-units.

block_dim

Pointer to the variable which will be set to the size of a block-unit.

DESCRIPTION

AMGX_vector_get_size retrieves the size of a **Vector** object in terms of number of entries, and the size of each entry.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

`AMGX_RC_BAD_PARAMETERS`

EXAMPLE

```
AMGX_vector_handle vector;
AMGX_vector_create(&vector, AMGX_mode_dDDI);
AMGX_vector_set_zero(vector, 100, 1);
int n, b;
AMGX_vector_get_size(vector, &n, &b);
```

HISTORY

AMGX_vector_get_size was introduced in API Version 1.

SEE ALSO

Vector Upload, *Vector Set Zero*

1.6.7 Vector Bind

NAME

AMGX_vector_bind - Create communication maps and partition info on a **Vector** by copying them from a **Matrix**.

SYNOPSIS

```
#include <amgx_c.h>
AMGX_RC AMGX_API AMGX_vector_bind(AMGX_vector_handle vec,
  AMGX_matrix_handle matrix);
```

PARAMETERS

vec

The **Vector** object for which communication maps and optional partition data are to be created.

matrix

The **Matrix** object which has communication maps and optional partition data already.

DESCRIPTION

AMGX_vector_bind defines communication maps between a local vector partition and one or more remote vector partitions by copying them from a **Matrix** which has communication maps on it already.

Because the **AMGX_vector_upload** routine may apply reordering or other optimizations based on communication patterns, this routine must be called *before* **AMGX_vector_upload**. Calling it *after* **AMGX_vector_upload** will result in the **Vector** to not actually have any attached communication information. In other words, a call to **AMGX_vector_bind** only records communication and partition information, but this information isn't actually uploaded and applied to the **Vector** until a subsequent call to **AMGX_vector_upload**. From a functional point of view, **AMGX_vector_set_zero** behaves identically to **AMGX_vector_upload** where the uploaded data is all zeroes, so the same semantics apply.

Once the communication maps and partition information have been attached to a **Vector** via **AMGX_vector_bind**, they do not need to be attached again before subsequent calls to **AMGX_vector_upload**, provided that the size of the **Vector** is unchanged. If the size does change, then new communication maps and partition information must be attached first via **AMGX_vector_bind**.

In the distributed setting, communications maps must be attached to a **Vector** via **AMGX_vector_bind** or **AMGX_read_system_distributed** before the **Vector** is passed to **AMGX_solver_solve** as either a right-hand side or solution vector.

RETURN VALUES

Relevant return values:

`AMGX_RC_OK`

AMGX_RC_BAD_MODE

AMGX_RC_NO_MEMORY

AMGX_RC_UNKNOWN

EXAMPLE

```
int n = ... // # of rows in the matrix
int nnz = ... // nnz in the matrix
float matrix_data[] = ...;
float matrix_cols[] = ...;
int   matrix_rows[] = ...;

float rhs_data[] = ...;

AMGX_matrix_handle mtx;
AMGX_vector_handle rhs, sol;
AMGX_resources_handle rsrc;

// single threaded using a single GPU
int devices[] = {0};
AMGX_resources_create(rsrc, config, NULL, 2, devices);

AMGX_matrix_create(mtx, rsrc, AMGX_mode_dFFI);
AMGX_vector_create(rhs, rsrc, AMGX_mode_dFFI);
AMGX_vector_create(sol, rsrc, AMGX_mode_dFFI);

AMGX_vector_bind(rhs, mtx);
AMGX_vector_bind(sol, mtx);

AMGX_matrix_upload_all(
  mtx, n, nnz, 1, 1, matrix_rows, matrix_cols, matrix_data, 0);
AMGX_vector_upload(rhs, n, 1, rhs_data);
AMGX_vector_set_zero(sol, n, 1);
// mtx, rhs, and sol can now be used by a Solver
```

HISTORY

AMGX_vector_bind was introduced in API Version 2.

SEE ALSO

Matrix Comm From Maps, *Matrix Comm From Maps One Ring*, *Vector Upload*, *Read System Distributed*

Chapter 2

Algorithm Guide

OVERVIEW

The AMGX library provides access to a number of different algorithms for solving sparse linear systems of equations. Deciding which one to use for a particular problem is a complex question, often requiring a deep knowledge of the application domain and physical properties of the system to be solved.

This guide describes the algorithms available via AMGX along with options and parameters that can be used to adjust these algorithms. Because AMGX provides a flexible pluging-based architecture, many different algorithms can be "mixed and matched" to create a wide variety of solvers. For example, any smoother can be paired with any AMG solution algorithm.

AMGX supports block or scalar systems, diagonal entries stored separately or with the off-diagonal entries, single or dobule precision floating point, and host or device-based storage. Since each of these combinations may require a different implementation of some routine, not all settings and algorithms are compatible with all storage and layouts. Attempting to use a certain algorithm with an incompatible storage type will result in a run-time error. Where possible, this guide notes these limitations.

It is also possible to implement custom algorithms and solvers via AMGX's plugin interface. A description of this interface is beyond the scope of the API reference.

Config Syntax

Resources Settings

General Settings

Multigrid Settings

Classical Algebraic Multigrid

Aggregation Multigrid

Krylov Solvers

Smoothers

Cycles

Nested Solvers

Examples

2.1 Config Syntax

OVERVIEW

The Config format is a text reprentation of a **Config** object. Config declarations may be stored as text files and loaded into a **Config** object via **AMGX_config_create_from_file**, or passed to a **Config** object from memory via **AMGX_config_create**. In either case, the format is the same.

Config files are a series of key-value pairs that define parameters, algorithms, and other settings which determine the behavior of a solver. Config files allow for a simple scoping mechanism which can be used to recursively build complex algorithms from simple building blocks. Because of the flexible nature of the file format, new parameters can be added and registered via plugins and passed through the rest of the system, allowing for extensible functionality with user control.

Comments can be included at any point. A comment line begins with the # character, like this:

```
# this is a comment
```

The first non-comment line should be a version specifier, which allows for the format to be backward compatible. The version is set just like any key-value pair:

```
config_version=2
```

It is recommeneded for the user to specify and use **config_version** 2. For example, nested solvers are only supported in **config_version** 2 and later. Note that file versions are backwards compatible, so a version of the AMGX library capable of reading version 2 will also be capable of reading version 1. Also, note that if the **config_version** is not specified, it is assumed to be 1.

The Config file is a list of settings, specied as key-value pairs. Keys are described in this document, and associated with each key is either a default value if it is left unset, or the user-provided value if it is set. Values can be different types, including floating point, strings, integer values, or a special **Solver** type.

Values of type string, float, or int are specified as

```
key=value
```

or

```
solver_name:key=value
```

with one setting per line. Scopes, described below, allow for nested structures, such as a Krylov solver preconditioned by an AMG solver, which uses an ILU smoother.

Any object which iteratively modifies a solution vector is categorized as a **Solver**. For example, all smoothers, Krylov solvers, AMG solver, or other iterative algorithms are considered **Solver**, and they can be used interchangably.

The syntax for specifying a value of type **Solver** is

```
key(solver_name)=value
```

where **value** will be the **Solver** type, and **solver_name** will be the user-given name of this **Solver** instance, used to reference the scope when defining a nested structure. Parameters on the solver can then be set using the `solver_name:key=value` syntax shown above.

Many **Solver** objects themselves can reference another **Solver** as one a setting. For exaple, the **FGMRES** solver has a parameter **preconditioner**, which can reference any other **Solver** object. For example, creating an FMGRES solver preconditioned by a Gauss-Seidel smoother can be written:

```
solver(my_krylov_solver)=FGMRES
my_krylov_solver:preconditioner(my_precond)=MULTICOLOR_GS
```

AMGX uses one-level scoping, so parameters set in the global scope (without a **solver_name** prefix) will be inherited by all *Solver* objects, unless that parameter is overriden on a particular **Solver**. Furthermore, scope names are not nested, so for example, to specify that the **MULTICOLOR_GS** smoother should be applied symmetrically, you would write

```
my_precond:symmetric_gs=1
```

Rather than the nested (incorrect)

```
my_krylov_solver:my_precond:symmetric_gs=1
```

In this case, you could also write simply

```
symmetric_gs=1
```

in the global scope, and that would be inherited by all **Solver** objects (but ignored by any **Solver** that doesn't care about this parameter, which is currently all **Solvers** except for **MULTICOLOR_GS**).

HISTORY

Comments and nested solvers are only supported in **config_version** 2. Features specific to **config_version** 2 can only be read by *AMGX* with API version 1.1 and later.

SEE ALSO

Config, *Config Create*, *Config Create From File*

2.2 Resources Settings

OVERVIEW

All **Resources** objects can be controlled via the following settings.

PARAMETERS

communicator ***(string: MPI)***

The type of communicator to be used. By default, inter-process communication will use MPI, and the **comm** parameter is assumed to be of type `MPI_Comm`. **MPI_DIRECT** is also allowed, and assumes that the installed version if MPI support GPUDirect. If the **comm** parameter is NULL, this value will be ignored.

min_rows_latency_hiding ***(int: -1)***

Latency hiding is an option during halo exchange, where the communication pattern will attempt to overlap the sending and receiving on halo data with computation on non-halo data. For large numbers of rows, this can be a good idea. However, it has overhead and enforces that the halo rows are processed with a serial dependency on the non-halo rows, reducing the total amount of available parallelism. This setting specifies the number of matrix rows below which latency hiding will be disabled - that is, any levels in the multigrid hierarchy with fewer than **min_rows_latency_hiding** rows will not use latency hiding. Setting this value to -1 disables latency hiding entirely.

matrix_consolidation_lower_threshold ***(int: 0)***

Average number of rows at which matrices from different processes must be merged. Default is no merging.

matrix_consolidation_upper_threshold ***(int: 1000)***

Average number of rows that merged matrices from different processes should have. Default is no merging.

fine_level_consolidation ***(int: 0)***

Flag that enables or disables (default) fine level consolidation. The consolidation can only happen once, in other words, if the fine level consolidation is enabled then the coarse level consolidation will be disabled.

amg_consolidation_flag ***(int: 0)***

Flag that enables or disables (default) fine level consolidation in classical algebraic mutigrid solvers. The consolidation can only happen once, in other words, if the fine level consolidation is enabled then the coarse level consolidation will be disabled. The consolidation thresholds are automatically computed at runime (default), it is also possible to set them manually using **matrix_consolidation_lower_threshold** and **matrix_consolidation_upper_threshold**.

LIMITATIONS

None.

SEE ALSO

2.3 General Settings

OVERVIEW

All **Solver** objects can be controlled via basic settings such as the convergence criteria. In the case of nested solvers, these settings can apply to a specific solver via scoping, or generically to all solvers via the global scope.

PARAMETERS

convergence ***(string: ABSOLUTE)***

The type of convergence check that will be used. **ABSOLUTE** tests whether the norm of the residual is less than **tolerance**. **RELATIVE_INI_CORE** tests whether the ratio of norm of the residual has been reduced by **tolerance** relative to the initial norm of the residual. In a block system, each equation will be tested independently - in other words, the residual is treated as n vectors for an n-by-n block system. Convergence for the entire system is only achieved when all residual values are considered converged.

determinism_flag ***(int: 0)***

Since AMGX often relies on randomized algorithms, the results may vary from one run to another. When this value is set to 1, the algorithms will be tuned such that the results should be deterministic and repeatable. This typically results in a small performance penalty, on the order of 15%.

max_iters ***(int: 100)***

The maximum number of iterations before a solver will exit. Setting this to 1, for example, means that only a single iteration of the solver will be applied, regardless of any convergence test. If the convergence test succeeds before **max_iters** are executed, the solver will exit.

monitor_residual ***(int: 0)***

When this flag is set to 0, the convergence check will ignore the residual, and will simply run until **max_iters** is reached.

norm ***(string: L2)***

The type of norm used to test for convergence. Supported values are **L1**, **L2**, abd **LMAX**. **LMAX** is not currently supported for block systems.

solver ***(solver: AMG)***

There is one unique top level solver set at the global scope, which will be used during **AMGX_solver_setup** and **AMGX_solver_solve**. Since any **Solver** type may be used interchangeably, this can be any of the listed smoothers, Krylov solvers or algebraic multigrid methods. For example, allowable options include algebraic multigrid **AMG**, Krylov solvers **PCG**, **PBICGSTAB** and **FGMRES**, as well as smoothers **BLOCK_JACOBI** and **MULTICOLOR_DILU**.

tolerance ***(double: 1e-12)***

The tolerance used during a convergence check. The units are whatever is specified via the **norm** parameter, and the value that is compared depends on the **convergence** parameter.

print_solve_stats ***(int: 0)***

Report statistics about the solve, such as the memory, rate of convergence and norm of the residual through iterations.

print_grid_stats ***(int: 0)***

Report statistics about the multigrid hierarchy, such as the memory, number of rows per level and sparsity of the matrices.

obtain_timings ***(int: 0)***

Report the setup, solve and total time taken to obtain the solution.

LIMITATIONS

None.

SEE ALSO

2.4 Multigrid Settings

OVERVIEW

Any **Solver** of type **AMG** will create an Algebraic Multigrid solver. The specific class of AMG algorithm can be set via the **algorithm** parameter. However, many parameters are common to all AMG algorithms. This includes controls for cycling strategy, pre and post sweeps, and other aspects of hierarchy construction.

PARAMETERS

algorithm ***(string: CLASSICAL)***

Select which AMG algorithm to use. Currently this can be either **CLASSICAL** or **AGGREGATION**. See *Classical Algebraic Multigrid* and *Aggregation Multigrid* for more details.

cycle ***(string: V)***

The type of cycle to use. Allowable options are **V**, **W**, **F**, **CG** and **CGF**. See *Cycles* for more information on the various cycle types.

smoother ***(solver: BLOCK_JACOBI)***

Choose the smoother to use. Note that the smoother is a **Solver** object, and therefore it can be named and configured using any of the listed smoothers, Krylov solvers or algebraic multigrid methods. For example, allowable options include algebraic multigrid **AMG**, Krylov solvers **PCG**, **PBICGSTAB** and **FGMRES**, smoothers **BLOCK_JACOBI** and **MULTICOLOR_DILU**, as well as none indicated by **NOSOLVER**. The smoother object will ignore any settings for **max_iters**, instead relying on **presweeps**, **postsweeps**, and **coarsest_sweeps** to control iteration count. Also, by default the smoother object's **monitor_residual** value will be set to 0.

postsweeps ***(int: 1)***

The number of applications of the smoother to be applied after coarse correction. Since the smoother is a *Solver* object, it can be configured to perform a convergence check rather than rely on the **postsweeps** parameter. However, any setting for **max_iters** on the smoother object will be ignored, and **postsweeps** will be used instead.

presweeps ***(int: 1)***

The number of applications of the smoother to be applied before coarse correction. Since the smoother is a **Solver** object, it can be configured to perform a convergence check rather than rely on the **presweeps** parameter. However, any setting for **max_iters** on the smoother object will be ignored, and **presweeps** will be used instead.

max_levels ***(int: 100)***

Indicates the maximum number of levels to be constructed. Setting this to 1, for example, means that no coarse correction will be applied and a multi-level solver will behave equivalently to a smoother.

min_coarse_rows ***(int: 2)***

The minimum number of block rows in a level. If the level has less than **min_coarse_rows** it is considered to be the coarsest level and the algorithm stops generating new levels in the AMG hierarchy.

coarse_solver ***(solver: DENSE_LU_SOLVER)***

Specify a **Solver** to be used for the coarsest problem. When the **max_levels** is reached, the resulting system will be solved using this **Solver**. By default, this will be a dense LU solver.

dense_lu_num_rows ***(int: 128)***

The number of rows at which the AMG algorithm stops generating new levels in the AMG hierarchy and uses dense LU on the coarsest level. If this parameter is used, it will reset the **min_coarse_rows** parameter accordingly.

dense_lu_max_rows ***(int: 0)***

The number of rows above which the dense LU solver will not be triggered. If the matrix on the coarsest level is larger than this value, then a smoother will be used on the coarsest level instead of the dense LU. This parameter is disabled by default.

coarsest_sweeps ***(int: 2)***

Controls number of times the **coarse_solver** will be applied on the coarsest level. If **coarse_solver=NOSOLVER** then the smoother will be applied **coarsest_sweeps** times, otherwise the specified **Solver** will be applied **max_iters** times to solve the coarsest linear system. Notice that since the **coarse_solver** is a **Solver** object, it can be configured to perform a convergence check rather than rely on a fixed number of iterations to converge.

LIMITATIONS

None.

SEE ALSO

2.5 Classical Algebraic Multigrid

OVERVIEW

Classical Algebraic Multgrid (Classical AMG), also known as Ruge-Steuben AMG, or Selection AMG, is a family of methods where the coarse grid is formed by "selecting" fine points to be carried through the coarse grid. Interpolation weights are then assigned to each fine point indicating how the values at the fine points will be weighted when forming th value of the coarse point.

An overview of Classical AMG can be found in

Klaus Stueben. *Algebraic Multigrid (AMG): An Introduction with Applications*. Germany: GMD Forschungszentrum Informationstechnik, 1999.

The parallel implementation of Classical AMG in AMGX is based largely on *hypre*, in particular the following papers:

Hans De Sterck, Ulrike Meier Yang, Jeffrey J. Heys. "Reducing complexity in parallel algebraic multigrid preconditioners." *SIAM J. Matrix Anal. Appl* vol. 27 2006: 1019 - 1039.

Hans De Sterck, Robert D. Falgout, Joshua W. Nolting, Ulrike Meier Yang. "Distance-two interpolation for parallel algebraic multigrid." *Numerical Linear Algebra with Applications* vol. 15, issue 2-3 2008: 115-139.

Ulrike Meier Yang. "On long range interpolation operators for aggressive coarsening." *Numerical Linear Algebra with Applications* vol. 17, issue 2-3 2010: 432-472.

In general, Classical AMG is a good choice for more difficult linear systems. For properly discretized elliptic equations, it is theoretically optimal in the sense that the number of iterations required to reach a certain reduction in the norm of the residual should not grow as the problem size increases. In practice, we find that it achieves overall good robustness on a variety of problems.

The parallel implementation of Classical AMG tends to be slightly weaker than the serial implementation, due to slightly worse coupling between elements in the smoothers, and slightly worse average behavior of the parallel coarsening algorithm (PMIS).

Compared with Aggregation AMG, the stencil used for restriction tends to be considerably larger, and tends to have a less predictable non-zero structure. As a result, application of the prolongation and restriction operators is more expensive, and the coarse grids tend to be less sparse. Also, computation of the Galerkin projection to form a coarse operator is more expensive in terms of memory footprint and execution time. On the other hand, the larger restriction means that Classical AMG is usually more numerically robust.

PARAMETERS

interp_max_elements ***(int: -1)***

> When this value is greater than 0, it use only the interpolation points with the **interp_max_elements** largest weights in each interpolatory set. This is equivalent to the *Hypre* setting `BoomerAMGSetPMaxElmts`.

interp_truncation_factor ***(double: 1.1)***

> When this value is between 0 and 1, it will drop values from the interpolatory set if their interpolation weights are less than `interp_truncation_factor * max_row_element`, where **max_row_ele-**

ment is the highest interpolation weight in the set. This is equivalent to the *Hypre* setting `HYPRE BoomerAMGSetTruncFactor`.

interpolator ***(string: D1)***

Choose the method for setting interpolation weights. Allowable options are **D1** and **D2**. The **D1** algorithm uses an interpolatory set consisting of only directly connected vertices. **D2** uses an interpolatory set consisting of vertices that are indirectly connected as well ("distance 2" along the graph). **D2** is considerably more expensive during both setup and solve phases, but is often required to achieve convergence on difficult problems. For more details, see the descriptoin of the "ext+i" method in [Meier 2010].

max_row_sum ***(float: 1.1)***

When this value is between 0 and 1, consider all connections in a row to be *weak* if the sum of the off-diagonal elements is less than **max_row_sum** times the diagonal value. This is equivalent to the *Hypre* setting `BoomerAMGSetMaxRowSum`.

selector ***(string: PMIS)***

For Classical AMG, only the **PMIS** selector is allowed. Other selectors may be chosen for Aggregation AMG, but they are incompatible. The PMIS algorithm is described in [DeSterck *et al.* 2006].

strength ***(string: AHAT)***

Choose the strength of connection metric to use. Allowable options are **AHAT** and **ALL**. The **AHAT** algorithm is desribed in [DeSterck *et al.* 2006] Equation 3.1. **ALL** will not discard any edges.

strength_threshold ***(double: 0.25)***

Threshhold for the **AHAT** strength of connection algorithm. All edges with strength below this threshhold will be discarded.

LIMITATIONS

Classical AMG only supports scalar and does not currently support block matrices. Furthermore, only **PMIS** and **HMIS** selectors as well as **D1** and **D2** interpolators are currently supported.

SEE ALSO

Aggregation Multigrid

2.6 Aggregation Multigrid

OVERVIEW

Aggregation Multgrid is a family of methods where the coarse grid is formed by aggregating (averaging) values from multiple fine points to form a coarse point. In an aggregation scheme, the coarse points do not correspond directly to fine points, but rather each coarse point represents a set, or "aggregate," from the fine grid.

The method employed by AmgX is sometimes calls "plain aggregation" or "unsmoothed aggregation." It is slightly different from what has been described in literature, but a good starting point is either of these papers:

Yvan Notay. "An aggregation-based algebraic multigrid method." ***Electronic Transactions on Numerical Analysis*** **vol. 37 2010: 123-146.**

HwanHo Kim, Jinchao Xu, Ludmil Zikatanov. "A multigrid method based on graph matching for convection-diffusion equations." ***Numerical Linear Algebra with Applications*** **vol 10, issue 2-3 2003: 181-195.**

AmgX's parallel aggregation scheme is based on a *parallel graph matching* approach, which will result in different types of aggregates from a *parallel maximal independent set (PMIS)* approach employed by other parallel AMG libraries (such as *cusp*). Graph matching provides greater control over the size of each aggregate. Unsmoothed aggregation with graph matching results in coarse grids that have the same average fill-in per matrix row as the fine grid. This property means that Aggregation Multigrid tends to use less memory than Classical AMG.

As a trade-off for improved efficiency, unsmoothed aggregation tends to be numerically weaker. Unsmoothed aggregation methods are known to not be numerically optimal, in the sense that the number of iterations required to reach a certain relative reduction in the norm of the residual grows as the problem size increases. As explained in [Notay 2010], it is possible to make Aggregation AMG asymptotically optimal by using a more expensive cycling strategy such as a K-cycle. For many problems, it appears to be faster in practice to use a sub-optimal V-cycle at the expense of additional iterations.

In AmgX, the graph matching algorithm we use is iterative, where initially all vertices are considered as singletons, and in each step they are merged into pairs. Because at each step, the number of vertices merged into pairs may will be less than the remaining number of singleton vertices, it is possible that some vertices may be "left out" of the aggregation and remain as singletons. In this case, the coarse correction at these points will not provide any benefit. Therefore, there is a trade-off between the number of graph matching iterations and the general numerical strength of the Aggregation AMG solver.

PARAMETERS

max_matching_iterations ***(int: 15)***

The iterative graph matching algorithm will terminate when either a large enough percentage of vertices have been merged, or **max_matching_iterations** is reached. It may be helpful to raise this to a large number, such as 200, on especially difficult problems.

max_unassigned_percentage ***(double: 0.05)***

The iterative graph matching algorithm will terminate when fewer than **max_unassigned_percentage** of the vertices remain to be merged, or **max_matching_iterations** is reached. This number should be between 0 and 1.

selector ***(string: PMIS)***

Specify the algorithm used to select aggregates. While **PMIS** is the default value, it is not a valid choice with Aggregation AMG. Valid options are **SIZE_2**, **SIZE_4**, and **SIZE_8**. **SIZE_2** will mostly select aggregates of size 2, while **SIZE_4** and **SIZE_8** will mostly build aggregates of size 4 and 8, respectively.

LIMITATIONS

Currently block sizes ranging from 1 to 5 are supported by the Aggregation AMG solver.

SEE ALSO

Classical Algebraic Multigrid

2.7 Krylov Solvers

OVERVIEW

AMGX supports using the AMG solver as a preconditioner for an iterative Krylov solver. Since the notion of a **Solver** is flexible, any of the Krylov solvers listed below could be used as a smoother, or a coarse grid solver. For those which accept a preconditioner, they can be used as an outer solver preconditioned by any other smoother, multigrid solver, or even another Krylov solver.

For example, to use a FGMRES solver with an error tolerance of 0.01 and maximum of 40 iterations (using Krylov subspace of size 10 and performing 4 restarts) without any preconditioning specify:

```
solver(fgmres)=FGMRES
fgmres:tolerance=0.01
fgmres:max_iters=40
fgmres:gmres_n_restart=10
fgmres:preconditioner=NOSOLVER
```

A more typical case would be to use something like FGMRES preconditioned by an AMG solver. That can be specified as:

```
solver=FGMRES
fgmres:tolerance=0.01
fgmres:max_iters=40
fgmres:gmres_n_restart=10
preconditioner(amg_solver)=AMG
amg_solver:max_iters=1
amg_solver:cycle=V
```

Note that any **Solver** could be used as a preconditioner, including any of the smoothers.

TYPES

BICGSTAB

The **BICGSTAB** solver type implements the Bi-Conjugate Gradient Stabilized (BiCGStab) algorithm, without preconditioning. BiCGStab was described in

Van der Vorst, H. A. "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems." ***SIAM Journal on Scientific and Statistical Computing*** **13 (2) 1992: 631-644.**

BiCGStab is used to solve non-symmetric linear systems with minimal extra storage. Each iteration of BiCGStab requires roughly twice as many operations as one iteration of Conjugate Gradients. Unlike FGMRES, BiCGStab does not require storage of trailing Krylov vectors.

CG

The **CG** solver type implements the Conjugate Gradients algorithm, without preconditioning. Conjugate Gradient is designed to solve symmetric, positive-definite systems (all positive eigenvalues) with minimal storage.

FGMRES

The **FGMRES** solver type implements the Flexible Generalized Minimum Residual (FGMRES) algorithm, with preconditioning. FGMRES can be used to solve non-symmetric and even indefinite linear systems. The flexible variant, which allows for the effective preconditioning operator to vary between iterations, is described in

Saad, Y. "A Flexible Inner-Outer Preconditioned GMRES Algorithm." ***SIAM Journal on Scientific Computing*** **14 (2) 1993: 461-469.**

FGMRES requires the storage of some number of trailing Krylov vectors, so the storage cost of FGMRES can be high when using a large restart value. However, unlike BiCGStab, it requires only a single application of the preconditioner at each iteration.

PBICGSTAB

The **PBICGSTAB** solver type implements the Bi-Conjugate Gradient Stabilized (BiCGStab) algorithm with preconditioning [Van der Vorst 1992]. BiCGStab is used to solve non-symmetric linear systems with minimal extra storage. Each iteration of preconditioned BiCGStab applies the preconditioner twice, with the same matrix but two different right-hand sides. Unlike FGMRES, BiCGStab does not require storage of trailing Krylov vectors.

PCG

The **PCG** solver type implements the Conjugate Gradients algorithm with preconditioning. Preconditioned Conjugate Gradient is designed to solve symmetric, positive-definite systems (all positive eigenvalues) with minimal storage.

PCGF

The **PCGF** solver type implements the Flexible Conjugate Gradients algorithm with preconditioning. Flexible preconditioned Conjugate Gradients is designed to solve symmetric, positive-definite systems (all positive eigenvalues) when the application of the preconditioner may result in a different effective preconditioning matrix at each iteration. For this reason, **PCGF** is recommended to use as the Krylov solver with an AMG preconditioner, or any non-deterministic preconditioner.

PARAMETERS

gmres_n_restart ***(int: 20)***

Applies only to the **FGMRES** solver type. This sets the size of the Krylov space before a restart is applied. Since FGMRES stores all trailing Krylov vectors, the storage requirement of the FGMRES solver grows proportional to this value.

max_iters ***(int: 100)***

The maximum number of iterations before a solver will exit. Setting this to 1, for example, means that only a single iteration of the solver will be applied, regardless of any convergence test. If the convergence test succeeds before **max_iters** are executed, the solver will exit.

Also, in the context of **FGMRES** solver, this paramtere specifies the total number of iterations performed. For example, if **max_iters**=40 and **gmres_n_restart**=10, then 4 restarts will be performed.

preconditioner ***(solver: AMG)***

Sets the solver to be used as a preconditioner for *Solver* types allowing preconditioning. Note that the preconditioner is a **Solver** object, and therefore it can be named and configured using any of the listed smoothers, Krylov solvers, and algebraic multigrid methods. For example, allowable options include algebraic multigrid **AMG**, Krylov solvers **PCG**, **PBICGSTAB** and **FGMRES**, smoothers **BLOCK_JACOBI** and **MULTICOLOR_DILU**, as well as none indicated by **NOSOLVER**.

LIMITATIONS

When using a solver type that does not support preconditioning, any **preconditioner** settings will be ignored.

Selecting the appropriate Krylov solver to use for a given linear system often requires knowledge of the underlying linear system. Using an inappropriate choice may result in either lack of convergence, or possibly even divergence and NAN values appearing in the solution vector.

SEE ALSO

Smoothers

2.8 Smoothers

OVERVIEW

A smoother is a lightweight iterative solver which tends to quickly damp the high frequency error modes, while leaving the lower frequency error modes largely unaffected. AMGX supports a number of parallel smoothers. For example, the following can be used to configure an underrelaxation smoother on an AMG solver with 0.5 weight:

```
solver(my_solver) = AMG
my_solver:smoother(my_smoother) = JACOBI
my_smoother:relaxation_factor = 0.5
```

Note that the smoother in this case is actually a nested *Solver* object, separate from the top-level AMG solver. Therefore, setting the **relaxation_factor** value on the solver will not work, since parameters are only inherited from the global scope. In other words, the following will not behave as expected, since it will set the **relaxation_factor** parameter on the AMG object itself, rather than on the smoother:

```
my_solver:relaxation_factor = 0.5
```

TYPES

BLOCK_JACOBI

The Jacobi method is a highly parallel smoother that tends to be weaker than other smoothers. It is very cheap to setup and apply. Jacobi can be damped, which is also called SSOR. When applied to a block matrix, Jacobi will operate in block mode and exactly inverts the diagonal block. Although the smoother is called **BLOCK_JACOBI**, it can be applied to scalar matrices.

MULTICOLOR_DILU

The DILU (Diagonal Incomplete LU) smoother is a form of approximate Incomplete LU factorization. AMGX uses a graph coloring approach to parallelize the traditional DILU algorithm. DILU is weaker than ILU0, but it uses considerably less storage and is much cheaper to factorize. Relative to Jacobi or Gauss-Seidel, the setup cost for DILU can be expensive. When applied to a block matrix, DILU will operate in block mode, exactly inverting each small block as needed.

MULTICOLOR_GS

Gauss-Seidel is usually numerically stronger than Jacobi but weaker than DILU. AMGX uses graph coloring to parallelize Gauss-Seidel. Setup and application costs are typically in between Jacobi and DILU as well. When applied to a block matrix, Gauss-Seidel will operate in block mode, exactly inverting each small block as needed.

MULTICOLOR_ILU

ILU (Incomplete LU) is typically the most numerically powerful smoother. AMGX uses graph coloring to parallelize both the ILU factorization and solution phases. Unlike DILU, which only requires an additional vector of storage, ILU requires an entire matrix of factors. The size of the matrix depends on the **ilu_sparsity_level**, but may be quite high, especially for dense systems. When applied to a block matrix, ILU will operate in block mode, exactly inverting each small block as needed.

POLYNOMIAL

Polynomial smoothers are highly parallel, similar to Jacobi, but result in superior numerical performance due to more optimal weighting of each update to the iterate. The approach in AMGX is based on the method described in:

Kraus, J, Vassilevski, P, Zikatanov, L. "Polynomial of best uniform approximation to x^{-1} and smoothing in two-level methods." Retrieved May 2012. Available at http://arxiv.org/abs/1002.1859

It offers similar numerical performance as the more standard Chebychev polynomial approach, but requires less accurate bounds on the eigenvalues of the system matrix and is therefore cheaper to setup. We will sometimes refer to this as the KPZ polynomial.

PARAMETERS

coloring_level ***(int: 1)***

For the **MULTICOLOR** smoothers which require a graph coloring, **coloring_level** indicates the distance of the independent sets formed by each color. For example, when **coloring_level** is 1, each vertex of a given color will be at least distance 1 from all other vertices of the same color. When **coloring_level** is 2, each vertex will be at least distance 2 from all other vertices of the same color, and so on. A higher value will result in more colors and less available parallelism during the application of the smoother. When using the **MULTICOLOR_ILU** smoother, this value must be set to at least one higher than the **ilu_sparsity_level** setting.

matrix_coloring_scheme ***(scheme: MIN_MAX)***

The multi-coloring algorithm to be used to color the graph associated with the matrix. The allowed values are **PARALLEL_GREEDY** and **MIN_MAX** (*default*).

ilu_sparsity_level ***(int: 0)***

When using the **MULTICOLOR_ILU** solver, selects the level of fill-in to be generated. Setting this to 0 results in IL0, 1 results in ILU1, and so on. Note that both the storage requirements and the setup and solve time will increase considerably with this value. Roughly, ILU0 requires about half as much storage as the system matrix, ILU1 requires half as much storage as a *square* of the system matrix, and so on. When **ilu_sparsity_level** is set to a value N, the **coloring_level** value must be set to at least N+1. Currently only values of 0 and 1 are supported.

kpz_order ***(int: 3)***

When using the **POLYNOMIAL** smoother type, this is the order of the polynomial. Roughly, this corresponds to the number of iterations to be applied. So for example, if **pre_sweeps** is 2 and **kpz_order** is 3, that would be equivalent in computational cost to **pre_sweeps** of 6 using the **JACOBI** smoother.

max_uncolored_percentage ***(double: 0.15)***

For the **MULTICOLOR** smoothers which require a graph coloring, **max_uncolored_percentage** indicates the fraction of vertices that can be left "uncolored." Uncolored vertices will result in weaker numerical coupling during the smoother application. However, it is often faster to leave some fraction of the vertices uncolored because then both the setup and solve phases will require fewer global sychronization points. When **determinism_flag** is set, this value will be overriden to be 0.

relaxation_factor ***(double: 0.9)***

Amount to weight the smoother correction by. **relaxation_factor** can be used with any of the **MULTICOLOR_GS**, **JACOBI**, or **MULTICOLOR_DILU** smoothers.

symmetric_GS ***(int: 0)***

For the **MULTICOLOR_GS** smoother, whether the Gauss-Seidel sweeps should be applied symmetrically. One sweep of Symmetric Gauss-Seidel is roughly equivalent to two sweeps of normal Gauss-Seidel, in terms of both numerical and computational performance.

LIMITATIONS

The **MULTICOLOR_ILU** only support **ilu_sparsity_level** of 0 or 1.

SEE ALSO

Multigrid Settings

2.9 Cycles

OVERVIEW

Multigrid methods can use different cycling strategies to move from finer to coarser levels. The choice of optimal cycling stategy depends on the relative costs and numerical necessities of reducing the various error modes. On a GPU, the cost of a coarse grid is higher relative to a finer grid when compared with a CPU. Therefore, cycling strategies that are most efficient on a CPU will not necessarily be most efficient on a GPU. For example, F and W cycles are often the best choice on a CPU AMG code, while the basic V cycle seems to perform better in many cases on a GPU.

TYPES

CG

A form of Krylov or non-linear AMLI cycle, where the coarse correction is is applied multiple times, modulated via some polynomial. A **CG** cycle is equivalent to using the coarse correction to precondition Conjugate Gradients applied to the fine-level matrix. Formally, this should be applied to symmetric matrices only. Pre and post smooth steps are applied before and after the application of the coarse correction polynomial.

CGF

A form of Krylov or non-linear AMLI cycle, where the coarse correction is is applied multiple times, modulated via some polynomial. A **CG** cycle is equivalent to using the coarse correction to precondition Flexible Conjugate Gradients applied to the fine-level matrix. Formally, this should be applied to symmetric matrices only. Since the coarse correction will typically result in a different effective preconditioner from one iteration to the next, **CGF** will usually be more robust than **CG**. Pre and post smooth steps are applied before and after the application of the coarse correction polynomial.

F

An F cycle is a combination of a V and W cycle. It has convergence quality similar to a W cycle, but costs less.

V

The basic V cycle, with one coarse correction applied between the pre-smoother and post-smoother.

W

The basic W cycle, with two coarse corrections applied between the pre-smoother and post-smoother.

PARAMETERS

cycle_iters ***(int: 2)***

For the **CG** and **CGF** cycle types, controls the degree of the coarse correction polynomial. In terms of computational cost, setting **cycle_iters** to 1 is roughly equal to a **V** cycle, and 2 is roughly equal to a **W** cycle.

LIMITATIONS

None.

SEE ALSO

Multigrid Settings

2.10 Examples

Aggregation AMG

The following settings create an Aggregation solver with DILU smoother, with 1 pre and 1 post sweep. The solver will stop when the L2 norm has been reduced by 1000 from the initial norm.

```
config_version=2
algorithm=AGGREGATION
selector=ONE_PHASE_HANDSHAKING
cycle=V
smoother=MULTICOLOR_DILU
presweeps=1
postsweeps=1
coarse_solver=NOSOLVER
coarsest_sweeps=2
max_levels=1000
norm=L2
convergence=RELATIVE_INI
max_uncolored_percentage=0.15
max_iters=1000
monitor_residual=1
tolerance=0.001
print_solve_stats=1
print_grid_stats=1
obtain_timings=1
```

In this simple case, there is no need to scope the parameters on the nested solvers since all values will be inherited from the global scope. The following Config is equivalent, with everything scoped explicitly:

```
config_version=2
solver(top_level)=AMG
top_level:algorithm=AGGREGATION
top_level:selector=ONE_PHASE_HANDSHAKING
top_level:cycle=V
top_level:smoother(my_smoother)=MULTICOLOR_DILU
my_smoother:max_uncolored_percentage=0.15
top_level:presweeps=1
top_level:postsweeps=1
top_level:coarse_solver=NOSOLVER
top_level:coarsest_sweeps=2
top_level:max_levels=1000
top_level:norm=L2
top_level:convergence=RELATIVE_INI_CORE
top_level:max_iters=1000
top_level:monitor_residual=1
top_level:tolerance=0.001
top_level:print_solve_stats=1
top_level:print_grid_stats=1
top_level:obtain_timings=1
```

Classical AMG

The following creates a Classical AMG solver with an outer FGMRES Krylov solver. The AMG solver will run 2 V-cycle for each FGMRES step, without checking for convergence. The outer FGMRES solver will run until the L2 norm of the residual is descreased by 4 orders of magnitude.

```
config_version=2
solver=FGMRES
gmres_n_restart=20
max_iters=100
norm=L2
convergence=RELATIVE_INI_CORE
monitor_residual=1
tolerance=1e-4
preconditioner(amg_solver)=AMG
amg_solver:algorithm=CLASSICAL
amg_solver:max_iters=2
amg_solver:presweeps=1
amg_solver:postsweeps=1
amg_solver:cycle=V
print_solve_stats=1
print_grid_stats=1
obtain_timings=1
```

Chapter 3

Programming Guide

OVERVIEW

The AMGX library may be used in a variety of contexts, from simple single-threaded applications with single-GPU support, up to complex hybrid MPI/OpenMP parallel applications with complex mappings from CPU threads to GPU devices. The different usage cases can be categorized roughly by the cardinalities of the mapping from CPU thread to GPU -- one-to-one, many-to-one, one-to-many and many-to-many. The latter two cases are currently not supported. This programming guide explains how to use the AMGX API in these different cases.

For the simple cases where a single thread controls a single GPU, there is nothing required more than creating the objects and solvers, and executing the solution phase. When multiple MPI ranks are being used, it becomes necessary to tell AMGX how the columns from one matrix on one rank map to the rows on neighboring ranks. With this information, AMGX can then distribute the matrix across multiple GPUs (one per MPI rank), or *consolidate* the matrix which is spread across multiple MPI ranks into one large matrix on a single device. From the programmer's perspective, these two use cases are the same, although obvously they will have very different performance characteristics.

Finally, these different scenarios may be combined in the case of a complex application with multiple MPI ranks spread across multiple nodes, each node with one or more GPUs, possibly using OpenMP for intra-node parallelism. AMGX may consolidate the matrices from multiple ranks onto a single GPU. With hybrid parallelism (OpenMPI within a node, MPI across nodes), a single OpenMP thread will be nominated to control the AMGX library. It must provide communication mappings from its matrix to the matrices on other MPI ranks.

These case are explained in detail in the following sections.

Single Thread Single GPU

Multi Thread OpenMP

Single Thread MPI

Multi Thread Hybrid

3.1 Single Thread Single GPU

OVERVIEW

In the simplest case, a single CPU thread has a linear system in memory that must be solved on a single GPU. The **AMGX_resources_create_simple** method is designed for this case - it will assign the default GPU device to the host thread.

The application creates all of the necessary **Vector** and **Matrix** objects, initializes a **Solver**, uploads the information to AMGX, and initiates a setup and solve phase. Finally, the results are downloaded from the device.

EXAMPLE

All of the objects are initialized using a default **Resources**.

```
AMGX_matrix_handle matrix;
AMGX_vector_handle rhs;
AMGX_vector_handle soln;
AMGX_resources_handle rsrc;
AMGX_solver_handle solver;
AMGX_config_handle config;

AMGX_resources_create_simple(&rsrc);
AMGX_matrix_create(&matrix, rsrc, AMGX_mode_dDDI);
AMGX_vector_create(&rhs, rsrc, AMGX_mode_dDDI);
AMGX_vector_create(&soln, rsrc, AMGX_mode_dDDI);

AMGX_config_create(&config, ...);
AMGX_solver_create(&solver, rsrc, AMGX_mode_dDDI, config);
```

Next, data is uploaded from the application (or set, in the case of the solution vector which is initialized to all zeroes).

```
AMGX_matrix_upload_all(matrix, ...);
AMGX_vector_upload(rhs, ...);
AMGX_vector_set_zero(soln, ...);
```

Finally, the setup and solve phases are executed.

```
AMGX_solver_setup(solver, matrix);
// slight optimization to tell is that soln is all zeros
AMGX_solver_solve_with_0_initial_guess(solver, rhs, soln);
```

The result vector can be copied back into an application buffer on the host.

```
AMGX_vector_download(soln, ...);
```

Note that in a typical case, the matrix will either remain fixed and multiple right-hand sides will be used in succession, or else the matrix structure will be fixed but the coefficients will change. The latter case will happen, for example, inside a non-linear solver such as Newton iterations, when the mesh is fixed by the non-linear equations are linearized multiple times. AMGX support the case of a fixed matrix by calling **AMGX_solver_solve** multiple times like this:

```
while (still_iterating) {
  AMGX_vector_upload(rhs, ...); // upload new data
  AMGX_solver_solve_with_0_initial_guess(solver, rhs, soln);
  AMGX_vector_download(soln, ...);

  // rest of app simulation loop
  ...
}
```

In the non-linear case where the matrix structure is fixed, this must be modified slightly:

```
while (still_iterating) {

  AMGX_matrix_replace_coefficients(matrix, ...);
  AMGX_solver_setup(solver, matrix); // must be called again

  AMGX_vector_upload(rhs, ...); // upload new data
  AMGX_solver_solve_with_0_initial_guess(solver, rhs, soln);
  AMGX_vector_download(soln, ...);

  // rest of app simulation loop
  ...
}
```

Note that in all of these cases, the host buffers which are read from or copied to should be pinned via **AMGX_pin_memory**.

3.2 Multi Thread OpenMP

OVERVIEW

OpenMP parallelism can be used in the case of multiple CPU cores which have shared access to memory, for example on a multicore CPU socket, or even across multiple CPU sockets on the same system.

AMGX supports an OpenMP parallel application by treating it the same as a single-threaded application. The application must nominate a single *master* thread which is responsible for communicating with AMGX. In other words, AMGX has no internal notion of supporting OpenMP parallelism - the application must interact with the AMGX library from a single thread. Internally, AMGX may spawn multiple CPU threads to accelerate some of the host-side computations, but this will be done independent from how the application is multithreaded, and the internal threads do not interact with application threads used outside of AMGX in any way.

The primary implication of this decision is that the application must have a single linear system which it will pass to AMGX - rather than partitioning its matrix data across multiple buffers, it must operate on a single global set of buffers. Or at least, it must assemble a single global data structure itself before calling into AMGX. Because OpenMP is a shared memory model, this requirement is typically compatible with how data is arranged internally.

One benefit of making AMGX unaware of the internal details of a multithreaded shared memory application is that AMGX is by default agnostic as to the threading method used - OpenMP, pthreads, Intel TBB, or any other threading library will be compatible with AMGX as long as communication goes through a single master thread.

Furthermore, it is straighforward for a shared-memory parallel CPU application to utilize multiple GPUs. From the perspective of the master thread, the usage will be identical to the single GPU multiple GPU case. As in the single GPU case, the application is responsble for providing partition information about how to distribute the system between the GPUs.

EXAMPLE

Since the OpenMP parallel case is largely identical to the single threaded case, we will omit many of the details for brevity.

It is important to note that **Resources** object will be initialized the same as the single-threaded case. For multiple GPUs, for example, the **Resources** could be initialized like this:

```
int devices[] = {0,1};
int num_devices = 2;
AMGX_resources_create(&rsrc, config, NULL, num_devices, devices);
```

for the single-GPU case, the **AMGX_resources_create_simple** interface may be used.

Uploading data uses the same interface, which requires that the matrix and vector buffers are already assembled on the host.

```
AMGX_matrix_upload_all(matrix, ...);
AMGX_vector_upload(rhs, ...);
```

The main thing to note is that all of these routines should either be called from outside any OpenMP parallel sections, or should be called from only a single thread. For example:

```
#pragma omp parallel
{
  // parallel
}

// serial
AMGX_resources_create_simple(&rsrc);
AMGX_matrix_create(&matrix, rsrc, AMGX_mode_dDDI);
AMGX_vector_create(&rhs, rsrc, AMGX_mode_dDDI);
AMGX_vector_create(&soln, rsrc, AMGX_mode_dDDI);
...
AMGX_solver_solve(solver, rhs, soln);
AMGX_vector_download(soln, ...);

#pragma omp parallel
{
  // parallel
}
```

Alternately, this can be called from inside the OpenMP block like this:

```
int master_thread_id = 0; // nominate one thread only
#pragma omp parallel
{
  // parallel
  ...

  // master thread only
  if (omp_get_thread_id() == master_thread_id) {
    AMGX_resources_create_simple(&rsrc);
    AMGX_matrix_create(&matrix, rsrc, AMGX_mode_dDDI);
    AMGX_vector_create(&rhs, rsrc, AMGX_mode_dDDI);
    AMGX_vector_create(&soln, rsrc, AMGX_mode_dDDI);
    ...
    AMGX_solver_solve(solver, rhs, soln);
    AMGX_vector_download(soln, ...);
  }

  // parallel
  ...
}
```

3.3 Single Thread MPI

OVERVIEW

MPI is a distributed memory parallel model which can either be used within a single node or across multiple nodes on a cluster. Because the mapping from MPI ranks to nodes is so flexible, and the number of GPUs per node may be less than, equal to, or greater than the number of MPI ranks on that node, it is necessary for AMGX to support different cardinalities of the relationship between MPI ranks and GPUs.

The AMGX library supports any case when the number of MPI ranks is greater than or equal to the number of GPUs.

In addition to describing how MPI ranks relate to GPUs, it is necessary for the application to tell AMGX how the MPI ranks related to each other. It is assumed that a matrix is partitioned across the different ranks, where the column index on one rank maps to a row index on a different rank. These mappings must be provided from the perspective of each MPI rank to AMGX via either **AMGX_matrix_comm_from_maps** or **AMGX_matrix_comm_from_maps_one_ring**.

In the case that muliple MPI ranks on the same node share a single GPU, AMGX internally will perform a *consolidation* step inside **AMGX_matrix_upload_all**, where the matrix partitions are merged into a single large matrix. That is, rather than have multiple MPI ranks share the GPU independently of each other, AMGX notices that they are all using the same GPU, and consolidates their linear system partitions into one. From the application perspective this behavior is hidden, and each MPI rank talks to AMGX as if the matrix partitions were distributed across multiple different memory spaces, even though they are in fact using shared memory within the GPU to communicate.

EXAMPLE

When creating the **Resources** object in an MPI application, it is necessary to provide AMGX with an MPI communicator to use. It is **highly** recommended that the application create a copy of its own communicator to eliminate the possibility of AMGX's internal MPI messages conflicting with the application's MPI messages. Based on the cardinality of the MPI Rank to GPU mapping, there are multiple cases to consider. In the simple case of one GPU per MPI rank, the **Resources** is created like this:

```
MPI_Comm AMGX_MPI_Comm = ... // copy the MPI communicator
AMGX_config_handle rsrc_config;
AMGX_config_create(rsrc_config, "communicator=MPI");

// app must know how to provide a mapping
int device[] = {get_device_id_for_this_rank()};
int num_devices = 1;
AMGX_resources_create(&rsrc, config, AMGX_MPI_Comm, num_devices, devices);
```

The case of multiple MPI ranks sharing a GPU would be identical, except that the call to `get_device_id_for_this_rank` in the above example would return the same device id for multiple MPI ranks. In the simplest case that there is a single device (id 0) attached to each node, this would look like this:

```
int device[] = {0};
int num_devices = 1;
AMGX_resources_create(&rsrc, config, AMGX_MPI_Comm, num_devices, devices);
```

Now all MPI ranks on this node will share the same GPU.

When using MPI, it is necessary to provide the communication maps *before* **AMGX_matrix_upload_all** is called. For example, the following would be a typical usage. This sequence would be executed in parallel across all MPI ranks.

```
AMGX_matrix_handle matrix;
AMGX_vector_handle rhs;
AMGX_vector_handle soln;
AMGX_solver_handle solver;
AMGX_config_handle config;

AMGX_matrix_create(&matrix, rsrc, AMGX_mode_dDDI);
AMGX_vector_create(&rhs, rsrc, AMGX_mode_dDDI);
AMGX_vector_create(&soln, rsrc, AMGX_mode_dDDI);
AMGX_config_create(&config, ...);
AMGX_solver_create(&solver, rsrc, AMGX_mode_dDDI, config);

// set comm maps and
// then propagate to vectors
AMGX_matrix_comm_from_maps_one_ring(matrix, ...);
AMGX_vector_bind(rhs, matrix);
AMGX_vector_bind(soln, matrix);

// if multiple ranks share a GPU, this will perform internal consolidation
// (exept for classical algebraic multigrid where consolidation is done in AMGX_solver_setup).
AMGX_matrix_upload_all(matrix, ...);
AMGX_vector_upload(rhs, ...);
AMGX_vector_upload(soln, ...);  // upload starting point for iterations

// solve and get the result
AMGX_solver_setup(solver, matrix);
AMGX_solver_solve(solver, rhs, soln);
AMGX_vector_download(soln, ...);
```

Note that the communication maps and partition depend on the matrix structure, so if the structure changes, the communication maps and partition must be reset.

```
while (still_simulating) {
  // import new matrix structure
  AMGX_matrix_comm_from_maps_one_ring(matrix, ...);
  AMGX_vector_bind(rhs, matrix);
  AMGX_vector_bind(soln, matrix);

  // if multiple ranks share a GPU, this will perform internal consolidation
  // (exept for classical algebraic multigrid where consolidation is done in AMGX_solver_setup).
  AMGX_matrix_upload_all(matrix, ...);
  AMGX_vector_upload(rhs, ...);
  AMGX_vector_upload(soln, ...);  // upload starting point for iterations
```

```
  // solve it
  AMGX_solver_setup(solver, matrix);
  AMGX_solver_solve(solver, rhs, soln);
  AMGX_vector_download(soln, ...);

  // do other stuff
}
```

However, if only the coefficients change, but the matrix structure is fixed, it is not necessary to set the communication maps or partition vector again. Because initializing these internal structures incurs significant overhead, it is highly recommended to use the **AMGX_matrix_replace_coefficients** fast-path whenever possible.

```
AMGX_matrix_comm_from_maps_one_ring(matrix, ...);
AMGX_vector_bind(rhs, matrix);
AMGX_vector_bind(soln, matrix);

// if multiple ranks share a GPU, this will perform internal consolidation
// (exept for classical algebraic multigrid where consolidation is done in AMGX_solver_setup).
AMGX_matrix_upload_all(matrix, ...);
AMGX_vector_upload(rhs, ...);
AMGX_vector_upload(soln, ...);  // upload starting point for iterations

while (still_simulating) {
  // coefficients changed
  AMGX_matrix_replace_coefficients(matrix, ...);

  // setup and solve it
  AMGX_solver_setup(solver, matrix);
  AMGX_solver_solve(solver, rhs, soln);
  AMGX_vector_download(soln, ...);

  // do other stuff
}
```

3.4 Multi Thread Hybrid

OVERVIEW

In a hybrid parallel model, computation is distributed across many MPI ranks, and each MPI rank launches multiple threads (for example using OpenMP). There is shared memory parallelism across the threads within a rank, and distributed memory parallelism between different ranks. AMGX assumes that each MPI rank is connected to one or more GPUs.

Because AMGX has no notion of shared memory parallelism on the host, it is necessary for each MPI rank to nominate a single *master* thread which will handle all communication with the AMGX library. In order to specify how the global system is distributed across the MPI ranks, it is necessary to set communication maps on the **Matrix** which indicate how column indices on one rank map to row indices on neighboring ranks.

These requirements are identical to the MPI-only or OpenMP-only usage models, except that n a hybrid model, it is necessary to combine them.

EXAMPLE

The Hybrid MPI example is almost identical to the MPI example, except that it is important to remember all AMGX calls must only be executed by the master thread on each rank, or called from outside any OpenMP parallel region.

```
int master_thread_id = 0; // nominate one thread only
#pragma omp parallel
{
  // parallel
  ...

  // master thread only
  if (omp_get_thread_id() == master_thread_id) {

    std::vector<int> device_list = get_device_id_list_for_this_rank();
    int num_devices = device_list.size();
    AMGX_resources_create(&rsrc, config, AMGX_MPI_Comm,
      num_devices, devic_list.begin());

    // create matrix, vectors, and solver
    ...

    AMGX_matrix_comm_from_maps_one_ring(matrix, ...);
    AMGX_vector_bind(rhs, matrix);
    AMGX_vector_bind(soln, matrix);

    AMGX_matrix_upload_all(matrix, ...);
    AMGX_vector_upload(rhs, ...);
    AMGX_vector_upload(soln, ...);  // upload starting point for iterations
```

```
    // solve it
    AMGX_solver_setup(solver, matrix);
    AMGX_solver_solve(solver, rhs, soln);

    // get the result
    AMGX_vector_download(soln, ...);
  }

  // parallel
  ...
}
```

Chapter 4

Legacy API

OVERVIEW

The AMGX library legacy API that consists of routines starting with prefix NVAMG is deprecated.

In order to convert to the current API please rename prefix of the routines from NVAMG to AMGX.

Index

Aggregation Multigrid, 136
Algorithm Guide, 125
API Reference, 1

Classical Algebraic Multigrid, 134
Config, 46
Config Create, 47
Config Create From File, 49
Config Destroy, 53
Config Get Default Number Of Rings, 51
Config Syntax, 127
Cycles, 144

Examples, 146

Finalize, 4
Finalize Plugins, 7
Free System Maps One Ring, 38

General Settings, 130
Get API Version, 9
Get Build Info Strings, 12
Get Error String, 10

Initialize, 3
Initialize Plugins, 5
Install Signal Handler, 18

Krylov Solvers, 138

Legacy API, 159

Matrix, 81
Matrix Comm From Maps, 100
Matrix Comm From Maps One Ring, 104
Matrix Create, 83
Matrix Destroy, 85
Matrix Get Size, 98
Matrix Replace Coefficients, 95
Matrix Upload All, 87
Matrix Upload All Global, 91
Multi Thread Hybrid, 157
Multi Thread OpenMP, 152
Multigrid Settings, 132

Pin Memory, 14
Programming Guide, 149

Read System, 21
Read System Distributed, 24
Read System Global, 27
Read System Maps One Ring, 32
Register Print Callback, 20
Reset Signal Handler, 19
Resources, 55
Resources Create, 56
Resources Create Simple, 58
Resources Destroy, 59
Resources Settings, 129

Single Thread MPI, 154
Single Thread Single GPU, 150
Smoothers, 141
Solver, 61
Solver Create, 63
Solver Destroy, 65
Solver Get Iteration Residual, 77
Solver Get Iterations Number, 75
Solver Get Status, 79
Solver Setup, 67
Solver Solve, 72
Solver Solve With 0 Initial Guess, 69

Unpin Memory, 16
Utilities, 2

Vector, 108
Vector Bind, 122
Vector Create, 110
Vector Destroy, 112
Vector Download, 116
Vector Get Size, 120
Vector Set Zero, 118
Vector Upload, 114

Write System, 41
Write System Distributed, 43

Notice

OpenCL

Trademarks

Copyright