Algebraic Programming (ALP) Tutorial

July 23, 2025

1 Welcome to GitHub Pages generated from LaTeX Document!

This document was compiled from LaTeX source code using GitHub Actions and deployed to GitHub Pages.

Contents

2 Quick Start

This section explains how to install ALP on a Linux system and compile a simple example. ALP (Algebraic Programming) provides a C++17 library implementing the GraphBLAS interface for linear-algebra-based computations. To get started quickly:

2.1 Installation on Linux

1. Install prerequisites: Ensure you have a C++11 compatible compiler (e.g. `g++` 4.8.2 or later) with OpenMP support, CMake (>= 3.13) and GNU Make, plus development headers for libNUMA and POSIX threads. For example, on Debian/Ubuntu:

   ```
   sudo apt-get install build-essential libnuma-dev libpthread-stubs0-dev cmake
   ```

2. Obtain ALP: Download or clone the ALP repository (from the official GitHub). For instance:

   ```
   git clone https://github.com/Algebraic-Programming/ALP.git
   ```

   Then enter the repository directory.

3. Build ALP: Create a build directory and invoke the provided bootstrap script to configure the project with CMake, then compile and install:

   ```
   $ cd ALP && mkdir build && cd build
   $ ../bootstrap.sh --prefix=../install # configure the build
   $ make -j # compile the ALP library
   $ make install # install to ../install
   ```

   (You can choose a different installation prefix as needed.)

4. Set up environment: After installation, activate the ALP environment by sourcing the script setenv in the install directory:

   ```
   $ source ../install/bin/setenv
   ```

   This script updates paths to make ALP's compiler wrapper and libraries available.

5. Compile an example: ALP provides a compiler wrapper `grbcxx` to compile programs that use the ALP/GraphBLAS API. This wrapper automatically adds the correct include paths and links against the ALP library and its dependencies. For example, to compile the provided sp.cpp sample:

   ```
   $ grbcxx ../examples/sp.cpp -o sp_example
   ```

   By default this produces a sequential program; you can add the option `-b reference_omp` to use the OpenMP parallel backend (shared-memory parallelism). The wrapper `grbcxx` accepts other backends as well (e.g. `-b hybrid` for distributed memory).

6. Run the program: Use the provided runner `grbrun` to execute the compiled binary. For a simple shared-memory program, running with `grbrun` is similar to using `./program` directly. For example:

   ```
   $ grbrun ./sp_example
   ```

   (The `grbrun` tool is more relevant when using distributed backends or controlling the execution environment; for basic usage, the program can also be run directly.)

After these steps, ALP is installed and you are ready to develop ALP-based programs. In the next sections we introduce core ALP concepts and walk through a simple example program.

3 Introduction to ALP Concepts

ALP exposes a programming model similar to the GraphBLAS standard, using algebraic containers (vectors, matrices, etc.) and algebraic operations on those containers. This section covers the basic data structures, the algebraic structures (semirings) that define how arithmetic is done, and key primitive operations (such as matrix-vector multiply and element-wise operations).

3.1 Vectors and Matrices in ALP

The primary container types in ALP are `grb::Vector<T>` and `grb::Matrix<T>`, defined in the `grb` namespace. These are templated on a value type `T`, which is the type of elements stored. Both vectors and matrices can be sparse, meaning they efficiently represent and operate on mostly-zero data by storing only nonzero elements webspace.science.uu.nl . For example, one can declare a vector of length 100000 and a 150000×100000 matrix as:

```
grb::Vector<double> x(100000), y(150000);
grb::Matrix<void> A(150000, 100000);
```

In this snippet, x and y are vectors of type double. The matrix A is declared with type void, which in ALP means it holds only the pattern of nonzero positions (no numeric values). Typically, one would use a numeric type (e.g. double) for matrix values; a void matrix is a special case where existence of an entry is all that matters (useful for boolean or unweighted graphs).

By default, new vectors/matrices start empty (with no stored elements). You can query properties like length or dimensions via `grb::size(vector)` for vector length, `grb::nrows(matrix)` and `grb::ncols(matrix)` for matrix dimensions, and `grb::nnz(container)` for the number of stored nonzero elements.

3.1.1 Exercise: Allocating Vectors and Matrices in ALP

Write a C++ program that uses ALP to allocate two vectors and one matrix as follows:

- A `grb::Vector<double>` x of length 100, with initial capacity 100.
- A `grb::Vector<double>` y of length 1000, with initial capacity 100.
- A `grb::Matrix<double>` A of size (100 × 1000), with initial capacity 100.

Make sure to include the necessary ALP headers, initialize the ALP context, and set the capacities via `resize`.

```
#include <iostream>
#include <graphblas.hpp>

int main() {
    // 1) Initialize ALP (using the sequential reference backend)
    grb::init< grb::reference >();

    // 2) Allocate vector x of length 100
    grb::Vector< double, grb::reference > x( 100 );
    grb::resize( x, 100 ); // Set initial capacity of x to 100 nonzeros

    // 3) Allocate vector y of length 1000
    grb::Vector< double, grb::reference > y( 1000 );
    grb::resize( y, 100 ); // Set initial capacity of y to 100 nonzeros

    // 4) Allocate matrix A of size 100 x 1000
    grb::Matrix< double, grb::reference > A( 100, 1000 );
    grb::resize( A, 100 );  // Set initial capacity of A to 100 nonzeros

    // 5) Print the capacities to verify
    std::cout << "Capacity of x: " << grb::capacity( x ) << std::endl;
    std::cout << "Capacity of y: " << grb::capacity( y ) << std::endl;
    std::cout << "Capacity of A: " << grb::capacity( A ) << std::endl;

    // 6) Finalize ALP
    grb::finalize();

    return 0;
}
```

When you run this program, ALP will print informational messages about initialization and finalization, and you will see lines reporting each container's capacity. In particular, you should observe output similar to:

```
Info: grb::init (reference) called.
Capacity of x: 100
Capacity of y: 1000
Capacity of A: 1000
Info: grb::finalize (reference) called.
```

3.2 Semirings and Algebraic Operations

A key feature of GraphBLAS (and ALP) is that operations are defined over semirings rather than just the conventional arithmetic operations. A semiring consists of a pair of operations (an "addition" and a "multiplication") along with their identity elements, which generalize the standard arithmetic (+ and ×). GraphBLAS allows using different semirings to, for example, perform computations like shortest paths or logical operations by substituting the plus or times operations with min, max, logical OR/AND, etc. In GraphBLAS, matrix multiplication is defined in terms of a semiring: the "add" operation is used to accumulate results, and the "multiply" operation is used when combining elements. ALP lets you define and use custom **semirings** by specifying:

- **A binary monoid:** an associative, commutative "addition" operation with an identity element. Examples:
    - (+, 0) — the usual addition over numbers
    - (`min`, +∞) — useful for computing minima

- **A binary multiplicative operator:** a second operation (not necessarily arithmetic multiplication), with its own identity element. Examples:
  - (⋆, 1) — standard multiplication
  - (AND, true) — logical semiring for Boolean algebra

A semiring is a combination of a multiplicative operator and an additive monoid. Many common semirings are provided or can be constructed. For instance, the plus-times semiring uses standard addition as the accumulation (monoid) and multiplication as the combination operator – this yields ordinary linear algebra over real numbers. One can also define a min-plus semiring (useful for shortest path algorithms, where "addition" is min and "multiplication" is numeric addition). ALP's design allows an "almost unlimited variety of operators and types" in semirings.

In code, ALP provides templates to construct these. For example, one can define:

```cpp
using Add = grb::operators::add<double>;
using AddMonoid = grb::Monoid<Add, grb::identities::zero>;
using Mul = grb::operators::mul<double>;
using PlusTimes = grb::Semiring<Mul, AddMonoid>;
PlusTimes plusTimesSemiring;
```

Here we built the plus-times semiring for double: we use the provided addition operator and its identity (zero) to make a monoid, then combine it with the multiply operator to form a semiring. ALP comes with a library of predefined operator functors (in grb::operators) and identities (in grb::identities) for common types. You can also define custom functor structs if needed. In many cases, using the standard plusTimesSemiring (or simply passing operators/monoids directly to functions) is sufficient for basic algorithms.

3.3 Primitive Operations (mxv, eWiseMul, dot, etc.)

Using the above containers and semirings, ALP provides a set of primitive functions in the grb namespace to manipulate the data. These functions are free functions (not class methods) and typically take the output container as the first parameter (by reference), followed by input containers and an operator or semiring specification. The most important primitives include:

grb::set – Assigns all elements of a container to a given value. For example, grb::set(x, 1.0) will set every entry of vector x to 1.0 (making all indices present with value 1.0). This is useful for initialization (if called on an empty vector, it will insert all indices with that value). There is also grb::setElement(container, value, index[, index2]) to set a single element: for a vector, you provide an index; for a matrix, a row and column. For example, grb::setElement(y, 3.0, n/2) sets $y_{n/2} = 3.0$.

grb::mxv – Perform matrix-vector multiplication on a semiring. The call grb::mxv(u, A, v, semiring) computes $u = A \otimes v$ (where $\otimes$ denotes matrix-vector multiply under the given semiring). For the plus-times semiring, this corresponds to the usual linear algebra operation $u_i = \sum_j A_{ij} \times v_j$ (summing with $+$ and multiplying with $\times$). The output vector u must be pre-allocated to the correct size (number of rows of A). By default, ALP's mxv adds into the output vector (as if doing $u+ = A \times v$). If you want to overwrite u instead of accumulate, you may need to explicitly set u to the identity element (e.g. zero) beforehand or use descriptors (advanced options) – but for most use cases, initializing u to 0 and then calling mxv is sufficient to compute $u = Ax$. For example, grb::mxv(y, A, x, plusTimesSemiring) will compute $y_i = \sum_j A_{ij} x_j$ using standard arithmetic (assuming y was zeroed initially).

grb::eWiseMul / grb::eWiseAdd (element-wise operations): These functions combine two containers element-by-element. For instance, grb::eWiseApply(z, x, y, op) will apply the binary operator op to each pair of elements x_i, y_i that exist (with matching index i), storing the result in z_i.

- **Intersection (eWiseMul):** If you use a plain binary operator (not a monoid) like multiplication or min, the result will include only entries where *both* input containers have entries. Essentially, missing values are ignored. For example:
  - grb::eWiseApply(z, x, x, grb::operators::min<double>()) sets $z_i = \min(x_i, x_i) = x_i$ for all indices i where x_i exists.
- **Union (eWiseAdd):** If you provide a *monoid* (which defines an identity), ALP will treat missing elements as identity values and operate over the union of indices. For example:

  - If y has a value at index k that x does not, `grb::eWiseApply(z, x, y, addMonoid)` yields $z_k = x_k + y_k = 0 + y_k = y_k$, where 0 is the identity of addition.

  Using an additive monoid results in an element-wise sum (union), while using a multiplicative operator gives an element-wise product (intersection).

In summary, to do element-wise multiplication (intersection), you might call: `grb::eWiseApply(z, x, y, grb::operators::mul<double>())`, which computes $z_i = x_i \times y_i$ for each i that has both x_i and y_i. To do an element-wise addition (union), you could call `grb::eWiseApply(z, x, y, addMonoid)`, where `addMonoid` encapsulates "+" with identity 0, resulting in $z_i = x_i + y_i$ for all indices that exist in either x or y. (If an index is missing in one operand, that operand contributes 0.)

grb::dot – Compute the dot product of two vectors. This is essentially a special case of a matrix-vector multiply or a reduce operation. ALP provides `grb::dot(result, u, v, semiring)` to compute a scalar result $= u^T \otimes v$ under a given semiring. For the standard plus-times semiring, `grb::dot(alpha, u, v, plusTimesSemiring)` will calculate $\alpha = \sum_i (u_i \times v_i)$ (i.e. the dot product of u and v). If you use a different monoid or operator, you can compute other pairwise reductions (for example, using a `min` monoid with logical multiplication could compute something like an "AND over all i" if that were needed). In most cases, you'll use dot with the default arithmetic semiring for inner products. The output `alpha` is a scalar (primitive type) passed by reference, which will be set to the resulting value.

grb::apply – Apply a unary operator or indexed operation to each element of a container. The function `grb::apply(z, x, op)` applies a given unary functor `op` to each element of vector (or matrix) `x`, writing the result into `z`. For example, if `op` is a functor that squares a number, `grb::apply(z, x, op)` would produce $z_i = op(x_i)$ for all stored elements of x. There are also forms of `apply` that use a binary operator with a scalar, effectively applying an affine operation. For instance, ALP could support `grb::apply(z, x, grb::operators::add<double>(), 5.0)` to add 5 to each element of x (if such an overload exists – conceptually, this would treat it as $z_i = x_i + 5$ for all i). In general, `apply` is like a map operation over all elements (it does not change the sparsity pattern — if x has an element at i, z will have an element at i after apply, unless filtered by a mask).

API usage notes: All the above operations require that output parameters be passed by reference, since they are modified in place (e.g., `y` in `grb::mxv(y, A, x, ...)` is updated with the result). Input objects are typically passed by const-reference. You should ensure that the output container is allocated with the correct size beforehand – ALP will not automatically resize vectors or matrices on operation calls if dimensions mismatch. If dimensions do not agree (e.g., you try to multiply an $m \times n$ matrix with a vector of length not n), the function will return an error code to indicate the misuse. In fact, most ALP primitives return a status code of type `grb::RC` (with `grb::SUCCESS` indicating success). For clarity, our code examples will omit explicit error handling, but in a real program you may check the returned code of each operation.

In the next section, we will put these concepts together in a concrete example.

4 Simple Example

To illustrate ALP usage, let's create a simple C++ program that:

- Initializes a small matrix A and vector x.
- Uses **set** and **setElement** to assign values.
- Performs a matrix-vector multiplication $y = A \times x$ (using **mxv** with the plus-times semiring).
- Computes a dot product $d = x \cdot x$ (using **dot**).
- Performs an element-wise multiplication $z = x * x$ (using **eWiseApply** with multiply operator).
- Prints the results.

Below is the code for this example, with commentary:

```
/*
 * example.cpp - Corrected minimal ALP (GraphBLAS) example.
 *
 * To compile (using the reference OpenMP backend):
 *    grbcxx -b reference_omp example.cpp -o example
 *
 * To run:
 *    grbrun ./example
 */

#include <cstdio>
#include <iostream>
#include <vector>
#include <utility>   // for std::pair
#include <array>

#include <graphblas.hpp>

using namespace grb;

// Indices and values for our sparse 3x3 matrix A:
//
//      A = [ 1   0   2 ]
//          [ 0   3   4 ]
//          [ 5   6   0 ]
//
// We store the nonzero entries via buildMatrixUnique.
static const size_t Iidx[6]    = { 0, 0, 1, 1, 2, 2 };  // row indices
static const size_t Jidx[6]    = { 0, 2, 1, 2, 0, 1 };  // column indices
static const double Avalues[6] = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 };

int main(int argc, char **argv) {
    (void)argc;
    (void)argv;
    std::printf("example (ALP/GraphBLAS) corrected API usage\n\n");

    //------------------------------
    // 1) Create a 3x3 sparse matrix A
    //------------------------------
    std::printf("Step 1: Constructing a 3x3 sparse matrix A.\n");
    Matrix<double> A(3, 3);
    // Reserve space for 6 nonzeros
    resize(A, 6);
    // Populate A from (Iidx,Jidx,Avalues)
    buildMatrixUnique(
        A,
        &(Iidx[0]),
        &(Jidx[0]),
        Avalues,
        /* nvals = */ 6,
        SEQUENTIAL
    );

    //------------------------------
    // 2) Create a 3-element vector x and initialize x = [1, 2, 3]^T
    //------------------------------
    std::printf("Step 2: Creating vector x = [1, 2, 3]^T.\n");
    Vector<double> x(3);
    set<descriptors::no_operation>(x, 0.0);           // zero-out
    setElement<descriptors::no_operation>(x, 1.0, 0); // x(0) = 1.0
    setElement<descriptors::no_operation>(x, 2.0, 1); // x(1) = 2.0
    setElement<descriptors::no_operation>(x, 3.0, 2); // x(2) = 3.0

    //------------------------------
    // 3) Create two result vectors y and z (dimension 3)
    //------------------------------
    Vector<double> y(3), z(3);
    set<descriptors::no_operation>(y, 0.0);
    set<descriptors::no_operation>(z, 0.0);

    //------------------------------
    // 4) Use the built-in "‘plusTimes’" semiring alias
    //      (add = plus, multiply = times, id-add = 0.0, id-mul = 1.0)
    //------------------------------
    auto plusTimes = grb::semirings::plusTimes<double>();

    //------------------------------
    // 5) Compute y = A·x  (matrix-vector multiply under plus-times)
    //------------------------------
    std::printf("Step 3: Computing y = A·x under plus-times semiring.\n");
    {
        RC rc = mxv<descriptors::no_operation>(y, A, x, plusTimes);
        if(rc != SUCCESS) {
            std::cerr << "Error: mxv(y,A,x) failed with code " << toString(rc) << "\n";
            return (int)rc;
        }
    }

    //------------------------------
    // 6) Compute z = x ⊙ x  (element-wise multiply) via eWiseApply with mul
    //------------------------------
    std::printf("Step 4: Computing z = x ⊙ x (element-wise multiply).\n");
    {
        RC rc = eWiseApply<descriptors::no_operation>(
            z, x, x,
            grb::operators::mul<double>()   // plain multiplication ⊙
        );
        if(rc != SUCCESS) {
            std::cerr << "Error: eWiseApply(z,x,x,mul) failed with code " << toString(rc)
    << "\n";
            return (int)rc;
        }
    }

    //------------------------------
    // 7) Compute dot_val = x^T·x  (dot-product under plus-times semiring)
    //------------------------------
    std::printf("Step 5: Computing dot_val = x^T·x under plus-times semiring.\n");
    double dot_val = 0.0;
    {
        RC rc = dot<descriptors::no_operation>(dot_val, x, x, plusTimes);
        if(rc != SUCCESS) {
            std::cerr << "Error: dot(x,x) failed with code " << toString(rc) << "\n";
            return (int)rc;
        }
    }

    //------------------------------
    // 8) Print x, y, z, and dot_val
    //    We reconstruct each full 3 - vector by filling an std::array<3,double>
    //------------------------------
    auto printVector = [&](const Vector<double> &v, const std::string &name) {
        // Initialize all entries to zero
        std::array<double,3> arr = { 0.0, 0.0, 0.0 };
        // Overwrite stored (nonzero) entries
        for(const auto &pair : v) {
            // pair.first = index, pair.second = value
            arr[pair.first] = pair.second;
        }
        // Print
        std::printf("%s = [ ", name.c_str());
        for(size_t i = 0; i < 3; ++i) {
            std::printf("%g", arr[i]);
            if(i + 1 < 3) std::printf(", ");
        }
        std::printf(" ]\n");
    };

    std::printf("\n-- Results --\n");
    printVector(x, "x");
    printVector(y, "y = A·x");
    printVector(z, "z = x ⊙ x");
    std::printf("dot(x,x) = %g\n\n", dot_val);

    return EXIT_SUCCESS;
}
```

Listing 1: Example program using ALP/GraphBLAS primitives in C++

In this program, we manually set up a 3×3 matrix A:

$$A = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 3 & 4 \\ 5 & 6 & 0 \end{pmatrix},$$

and a vector $x = [1, 2, 3]^T$. The code multiplies A by x, producing $y = A \times x$. Given the above A and x, the result should be:

$$y = \begin{pmatrix} 7 \\ 18 \\ 17 \end{pmatrix},$$

because $y_0 = 1 \cdot 1 + 0 \cdot 2 + 2 \cdot 3 = 7$, $y_1 = 0 \cdot 1 + 3 \cdot 2 + 4 \cdot 3 = 18$, $y_2 = 5 \cdot 1 + 6 \cdot 2 + 0 \cdot 3 = 17$.

We also compute the dot product $x \cdot x = 1^2 + 2^2 + 3^2 = 14$ and the element-wise product $z = x * x = [1, 4, 9]^T$. The program then extracts the results with `grb::extractElement` (to get values from the containers) and prints them. Running this program would produce output similar to:

```
y = [7, 18, 17]
x . x = 14
z (element-wise product of x with x) = [1, 4, 9]
```

This confirms that our ALP operations worked as expected. The code demonstrates setting values, performing an `mxv` multiplication, an element-wise multiply, and a dot product, covering several fundamental GraphBLAS operations.

When you run the example, the program first prints each step of the computation (building the matrix, creating the vector, etc.). In particular, you will see lines like

```
Step 1: Constructing a 3x3 sparse matrix A.
Step 2: Creating vector x = [1, 2, 3]^T.
Step 3: Computing y = A·x under plus-times semiring.
Step 4: Computing z = x ⊙ x (element-wise multiply).
Step 5: Computing dot_val = x^T·x under plus-times semiring.
```

After these messages, the program prints the final results for $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, and $\text{dot}(\mathbf{x}, \mathbf{x})$. In this particular example (with

$$A = \begin{bmatrix} 0 & 1 & 2 \\ 0 & 3 & 4 \\ 5 & 6 & 0 \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}),$$

one obtains

$$\mathbf{y} \; = \; A \otimes \mathbf{x} \; = \; \begin{bmatrix} 7 \\ 18 \\ 17 \end{bmatrix}, \quad \mathbf{z} \; = \; \mathbf{x} \odot \mathbf{x} \; = \; \begin{bmatrix} 1 \\ 4 \\ 9 \end{bmatrix}, \quad \text{dot}(\mathbf{x}, \mathbf{x}) \; = \; 14.$$

Hence, immediately after the step-headings, the console will display something like:

```
// Step 4: Computing z = x ⊙ x (element-wise multiply).
x = [ 1, 2, 3 ]
y = A·x = [ 7, 18, 17 ]
z = x ⊙ x = [ 1, 4, 9 ]
dot(x,x) = 14
```

5 Makefile and CMake Instructions

Finally, we provide guidance on compiling and running the above example in your own development environment. If you followed the installation steps and used grbcxx, compilation is straightforward. Here we outline two approaches: using the ALP wrapper scripts, and integrating ALP manually via a build system.

Using the ALP compiler wrapper

The simplest way to compile your ALP-based program is to use the provided wrapper. After sourcing the ALP environment (setenv script), the commands grbcxx and grbrun are available in your PATH. You can compile the example above by saving it (e.g. as example.cpp) and running:

```
$ grbcxx example.cpp -o example
```

This will automatically invoke g++ with the correct include directories and link against the ALP library and its required libraries (NUMA, pthread, etc.). By default, it uses the sequential backend. To enable parallel execution with OpenMP, add the flag -b reference_omp (for shared-memory parallelism). For instance:

```
$ grbcxx -b reference_omp example.cpp -o example
```

After compilation, run the program with:

```
$ grbrun ./example
```

(You can also run ./example directly for a non-distributed run; grbrun is mainly needed for orchestrating distributed runs or setting up the execution environment.)

Using a custom build (Make/CMake)

If you prefer to compile without the wrapper (for integration into an existing project or custom build system), you need to instruct your compiler to include ALP's headers and link against the ALP library and dependencies. The ALP installation (at the chosen -prefix) provides an include directory and a library directory.

For example, if ALP is installed in ../install as above, you could compile the program manually with:

```
$ g++ -std=c++17 example.cpp
-I../install/include -L../install/lib
-lgraphblas -lnuma -lpthread -lm -fopenmp -o example
```

Here we specify the include path for ALP headers and link with the ALP GraphBLAS library (assumed to be named libgraphblas) as well as libnuma, libpthread, libm (math), and OpenMP (the -fopenmp flag). These additional libraries are required by ALP (as noted in the install documentation). Using this command (or a corresponding Makefile rule) will produce the executable example.

If you are using CMake for your own project, you can integrate ALP as follows. There may not be an official CMake package for ALP, but you can use find_library or hard-code the path. For instance, in your CMakeLists.txt:

```
    cmake_minimum_required(VERSION 3.13)
    project(ALPExample CXX)
    find_package(OpenMP REQUIRED) # find OpenMP for -fopenmp
    Set ALP install paths (adjust ../install to your prefix)

    include_directories("../install/include")
    link_directories("../install/lib")

    add_executable(example example.cpp)
    target_link_libraries(example PRIVATE
        graphblas # ALP GraphBLAS library
        OpenMP::OpenMP_CXX # OpenMP (threading support)
        pthread numa m # Pthreads, NUMA, math libraries
    )
```

Listing 2: Example CMakeLists.txt for an ALP project

This will ensure the compiler knows where to find `graphblas.hpp` and links the required libraries. After configuring with CMake and building (via `make`), you can run the program as usual.

Note: If ALP provides a CMake package file, you could use `find_package` for ALP, but at the time of writing, linking manually as above is the general approach. Always make sure the library names and paths correspond to your installation. The ALP documentation mentions that programs should link with `-lnuma`, `-lpthread`, `-lm`, and OpenMP flags in addition to the ALP library.

This tutorial has introduced the fundamentals of using ALP/GraphBLAS in C++ on Linux, from installation to running a basic example. With ALP set up, you can explore more complex graph algorithms and linear algebra operations, confident that the library will handle parallelization and optimization under the hood. Happy coding!

6 Introduction to ALP and Transition Paths

Algebraic Programming (ALP) is a C++ framework for high-performance linear algebra that can auto-parallelize and auto-optimize your code. A key feature of ALP is its transition path APIs, which let you use ALP through standard interfaces without changing your existing code. In practice, ALP generates drop-in replacements for established linear algebra APIs. You simply re-link your program against ALP's libraries to get optimized performance (no code modifications needed). ALP v0.8 provides transition-path libraries for several standards, including the NIST Sparse BLAS and a CRS-based iterative solver interface (ALP/Solver). This means you can take an existing C/C++ program that uses a supported API and benefit from ALP's optimizations (such as vectorization and parallelism) just by linking with ALP's libraries [1].

One of these transition paths, and the focus of this tutorial, is ALP's **sparse Conjugate Gradient (CG) solver**. This CG solver accepts matrices in Compressed Row Storage (CRS) format (also known as CSR) and solves $Ax = b$ using an iterative method. Under the hood it leverages ALP's non-blocking execution model, which overlaps computations and memory operations for efficiency. From the user's perspective, however, the solver is accessed via a simple C-style API that feels synchronous. In this workshop, we'll learn how to use this CG solver interface step by step: from setting up ALP, to coding a solution for a small linear system, to building and running the program.

7 Setup: Installing ALP and Preparing to Use the Solver

This section explains how to install ALP on a Linux system and compile a simple example. ALP (Algebraic Programming) provides a C++17 library implementing the GraphBLAS interface for linear-algebra-based computations.

7.1 Installation on Linux

1. Install prerequisites: Ensure you have a C++11 compatible compiler (e.g. `g++` 4.8.2 or later) with OpenMP support, CMake (>= 3.13) and GNU Make, plus development headers for libNUMA and POSIX threads. For example, on Debian/Ubuntu:

   ```
   sudo apt-get install build-essential libnuma-dev libpthread-stubs0-dev cmake
   ```

2. Obtain ALP: Download or clone the ALP repository (from the official GitHub). For instance:

   ```
   git clone https://github.com/Algebraic-Programming/ALP.git
   ```

   Then enter the repository directory.

3. Build ALP: Create a build directory and invoke the provided bootstrap script to configure the project with CMake, then compile and install:

   ```
   $ cd ALP && mkdir build && cd build
   $ ../bootstrap.sh --prefix=../install # configure the build
   $ make -j # compile the ALP library
   $ make -j install # install to ../install
   $ source ../install/bin/setenv
   ```

   (You can choose a different installation prefix as needed.)

4. Set up environment: After installation, activate the ALP environment by sourcing the script setenv in the install directory:

   ```
   $ source ../install/bin/setenv
   ```

   This script updates paths to make ALP's compiler wrapper and libraries available.

5. Compile an example: ALP provides a compiler wrapper `grbcxx` to compile programs that use the ALP/GraphBLAS API. This wrapper automatically adds the correct include paths and links against the ALP library and its dependencies. For example, to compile the provided sp.cpp sample:

   ```
   $ grbcxx ../examples/sp.cpp -o sp_example
   ```

   By default this produces a sequential program; you can add the option `-b reference_omp` to use the OpenMP parallel backend (shared-memory parallelism). The wrapper `grbcxx` accepts other backends as well (e.g. `-b hybrid` for distributed memory).

6. Run the program: Use the provided runner `grbrun` to execute the compiled binary. For a simple shared-memory program, running with `grbrun` is similar to using `./program` directly. For example:

   ```
   $ grbrun ./sp_example
   ```

   (The `grbrun` tool is more relevant when using distributed backends or controlling the execution environment; for basic usage, the program can also be run directly.)

You can also specify a backend with the -b flag. For instance, -b reference builds a sequential version, while -b reference_omp enables ALP's shared-memory (OpenMP) parallel backend . If you built ALP with distributed-memory support, you might use -b hybrid or -b bsp1d for hybrid or MPI- nstyle backends. In those cases, you would run the program via grbrun (which handles launching multiple processes) – but for this tutorial, we will use a single-process, multi-threaded backend, so running the program normally is fine.

Direct linking option: If you prefer to compile with your usual compiler, you need to include ALP's headers and link against the ALP libraries manually. For the CG solver transition path, that typically means linking against the sparse solver library (e.g. libspsolver_shmem_parallel for the parallel version) and any core ALP libraries it depends on. For example, if ALP is installed in /opt/alp , you might compile with:

```
gcc -I/opt/alp/include -L/opt/alp/lib \
-lspsolver_shmem_parallel -lalp_cspblas_shmem_parallel \
my_program.c -o my_program
```

(ALP's documentation provides details on which libraries to link for each backend [3].) Using grbcxx is recommended for simplicity, but it's good to know what happens under the hood. Now that our environment is set up, let's look at the CG solver API.

8 Overview of ALP's Non-Blocking Sparse CG API

The ALP/Solver transition path provides a C-style interface for initializing and running a Conjugate Gradient solver. All functions are exposed via a header (e.g. solver.h in ALP's include directory) and use simple types like pointers and handles. The main functions in this API are:

- **sparse_cg_init(&handle, n, vals, cols, offs):** Initializes a CG solver instance. It allocates/assigns a solver context to the user-provided sparse_cg_handle_t (an opaque handle type defined by ALP). The matrix A is provided in CRS format by three arrays: vals (the nonzero values), cols (the column indices for each value), and offs (offsets in the vals array where each row begins). The dimension n (number of rows, which should equal number of columns for A) is also given. After this call, the handle represents an internal solver state with matrix A stored. Return: Typically returns 0 on success (and a non-zero error code on failure) [4].

- **sparse_cg_set_preconditioner(handle, func, data):** (Optional) Sets a preconditioner for the iterative solve. The user provides a function func that applies the preconditioner M^{-1} to a vector (i.e. solves $Mz = r$ for a given residual r), along with a user data pointer. ALP will call this func(z, r, data) in each CG iteration to precondition the residual. If you don't call this, the solver will default to no preconditioning (i.e. $M = I$). You can use this to plug in simple preconditioners (like Jacobi, with data holding the diagonal of A) or even advanced ones, without modifying the solver code. Return: 0 on success, or error code if the handle is invalid, etc.

- **sparse_cg_solve(handle, x, b):** Runs the CG iteration to solve $Ax = b$. Here b is the right-hand side vector (input), and x is the solution vector (output). You should allocate both of these arrays of length n beforehand. The solver will iterate until it converges to a solution within some default tolerance or until it reaches an iteration limit. On input, you may put an initial guess in x. If not, it's safe to initialize x to zero (the solver will start from $x_0 = 0$ by default in that case). Upon return, x will contain the approximate solution. Return: 0 if the solution converged (or still 0 if it ran the maximum iterations – specific error codes might indicate divergence or other issues in future versions).

- **sparse_cg_destroy(handle):** Destroys the solver instance and releases any resources associated with the given handle. After this, the handle becomes invalid. Always call this when you are done solving to avoid memory leaks. Return: 0 on success (and the handle pointer may be set to NULL or invalid after). This API is non-blocking in the sense that internally ALP may overlap operations (like sparse matrix-vector multiplications and vector updates) and use asynchronous execution for performance. However, the above functions themselves appear synchronous. For example, sparse_cg_solve will only return after the solve is complete (there's no separate "wait" call exposed in this C interface). The benefit of ALP's approach is that you, the developer, don't need to manage threads or message passing at all ALP's GraphBLAS engine handles parallelism behind the scenes. You just call these routines as you would any standard library. Now, let's put these functions into practice with a concrete example.

This API is non-blocking in the sense that internally ALP may overlap operations (like sparse matrix-vector multiplications and vector updates) and use asynchronous execution for performance. However, the above functions themselves appear synchronous. For example, sparse_cg_solve will only return after the solve is complete (there's no separate "wait" call exposed in this C interface). The benefit of ALP's approach is that you, the developer, don't need to manage threads or message passing at all. ALP's GraphBLAS engine handles parallelism behind the scenes. You just call these routines as you would any standard library. Now, let's put these functions into practice with a concrete example.

9 Example: Solving a Linear System with ALP's CG Solver

Suppose we want to solve a small system $Ax = b$ to familiarize ourselves with the CG interface. We will use the following 3×3 symmetric positive-definite matrix A:

$$A = \begin{pmatrix} 4 & 1 & 0 \\ 1 & 3 & -1 \\ 0 & -1 & 2 \end{pmatrix},$$

and we choose a right-hand side vector b such that the true solution is easy to verify. If we take the solution to be $x = (1,\ 2,\ 3)$, then $b = Ax$ can be calculated as:

$$b = \begin{pmatrix} 6 \ 4 \ 4 \end{pmatrix},$$

since $4 \cdot 1 + 1 \cdot 2 + 0 \cdot 3 = 6$, $1 \cdot 1 + 3 \cdot 2 + (-1) \cdot 3 = 4$, and $0 \cdot 1 + (-1) \cdot 2 + 2 \cdot 3 = 4$. Our goal is to see if the CG solver recovers $x = (1, 2, 3)$ from A and b.

We will hard-code A in CRS format (also called CSR: Compressed Sparse Row) for the solver. In CRS, the matrix is stored by rows, using parallel arrays for values and column indices, plus an offset index for where each row starts. For matrix A above:

- Nonzero values in row 0 are [4, 1] (at columns 0 and 1), in row 1 are [1, 3, -1] (at cols 0,1,2), and in row 2 are [-1, 2] (at cols 1,2). So the vals array will be {4, 1, 1, 3, -1, -1, 2} (concatenated by row).
- Corresponding column indices cols would be {0, 1, 0, 1, 2, 1, 2} aligned with each value.
- The offsets offs marking the start of each row's data in the vals array would be: row0 starts at index 0, row1 at index 2, row2 at index 5, and an extra final entry = 7 (total number of nonzeros). So offs = {0, 2, 5, 7}.

Below is a complete program using ALP's CG solver to solve for x. We include the necessary ALP header for the solver API, set up the matrix and vectors, call the API functions in order, and then print the result.

```c
#include <stdio.h>
#include <stdlib.h>

// Include ALPs solver API header
#include <transition/solver.h> // (path may vary based on installation)

int main(){
    // Define the 3x3 test matrix in CRS format
    const size_t n = 3;

    double A_vals[] = {
        4.0, 1.0, // row 0 values
        1.0, 3.0, -1.0, // row 1 values
        -1.0, 2.0 // row 2 values
    };

    int A_cols[] = {
        0, 1, // row 0 column indices
        0, 1, 2, // row 1 column indices
        1, 2 // row 2 column indices
    };

    int A_offs[] = { 0, 2, 5, 7 }; // row start offsets: 0,2,5 and total nnz=7

    // Right-hand side b and solution vector x
    double b[] = { 6.0, 4.0, 4.0 }; // b = A * [1,2,3]^T
    double x[] = { 0.0, 0.0, 0.0 }; // initial guess x=0 (will hold the solution)

    // Solver handle
    sparse_cg_handle_t handle;

    int err = sparse_cg_init_dii(&handle, n, A_vals, A_cols, A_offs);
    if (err != 0) {
        fprintf(stderr, "CG init failed with error %d\n", err);
        return EXIT_FAILURE;
    }

    // (Optional) set a preconditioner here if needed, e.g. Jacobi or others.
    // We skip this, so no preconditioner (effectively M = Identity).
    err = sparse_cg_solve_dii(handle, x, b);
    if (err != 0) {
        fprintf(stderr, "CG solve failed with error %d\n", err);
        // Destroy handle before exiting
        sparse_cg_destroy_dii(handle);
        return EXIT_FAILURE;
    }

    // Print the solution vector x
    printf("Solution x = [%.2f, %.2f, %.2f]\n", x[0], x[1], x[2]);

    // Clean up
    sparse_cg_destroy_dii(handle);

    return 0;
}
```

Listing 3: Example program using ALP's CG solver API

Let's break down what happens here:

- We included <graphblas/solver.h> (the exact include path might be alp/solver.h or similar depending on ALP's install, but typically it resides in the GraphBLAS include directory of ALP). This header defines the sparse_cg_* functions and the **sparse_cg_handle_t** type.
- We set up the matrix data in CRS format. For clarity, the values and indices are grouped by row in the

code. The offsets array {0,2,5,7} indicates: row0 uses vals[0..1], row1 uses vals[2..4] , row2 uses vals[5..6]. The matrix dimension n is 3.

- We prepare the vectors b and x. b is initialized to {6,4,4} as computed above. We initialize x to all zeros (as a starting guess). In a real scenario, you could start from a different guess, but zero is a common default.

- We create a **sparse_cg_handle_t** and call sparse_cg_init. This hands the matrix to ALP's solver. Under the hood, ALP will likely copy or reference this data and possibly analyze A for the CG algorithm. We check the return code err, if non-zero, we print an error and exit. (For example, an error might occur if n or the offsets are inconsistent. In our case, it should succeed with err $==$ 0.)

- We do not call **sparse_cg_set_preconditioner** in this example, which means the CG will run unpreconditioned. If we wanted to, we could implement a simple preconditioner. For instance, a Jacobi preconditioner would use the diagonal of A: we'd create an array with diag(A) $= [4, 3, 2]$ and a function to divide the residual by this diagonal. We would then call **sparse_cg_set_preconditioner(handle, my_prec_func, diag_data)**. For brevity, we skip this. ALP will just use the identity preconditioner by default (no acceleration).

- Next, we call **sparse_cg_solve(handle, x, b)**. ALP will iterate internally to solve $Ax = b$. When this function returns, x should contain the solution. We again check err. A non-zero code could indicate the solver failed to converge (though typically it would still return 0 and one would check convergence via a status or residual, ALP's API may evolve to provide more info). In our small case, it should converge in at most 3 iterations since A is 3×3.

- We print the resulting x. We expect to see something close to [1.00, 2.00, 3.00]. Because our matrix and b were consistent with an exact solution of $(1, 2, 3)$, the CG method should find that exactly (within floating-point rounding). You can compare this output with the known true solution to verify the solver worked correctly.

- Finally, we call **sparse_cg_destroy(handle)** to free ALP's resources for the solver. This is important especially for larger problems to avoid memory leaks. After this, we return from main.

Building and Running the Example

To compile the above code with ALP, we will use the direct linking option as discussed.

```
g++ example.cpp -o cg_demo \
  -I install/include \
  -L install/lib \
  -L install/lib/sequential \
  -Wl,-rpath,$PWD/install/lib/sequential \
  -lspsolver_shmem_parallel \
  -lalp_cspblas_shmem_parallel \
  -lgraphblas \
  -lalp_utils \
  -lksolver \
  -fopenmp \
  -lnuma \
  -no-pie
```

After a successful compile, you can run the program:

```
    ./cg_demo
```

It should output something like:

```
    Solution x = [1.00, 2.00, 3.00]
```

Congratulations! You've now written and executed a Conjugate Gradient solver using ALP's transition path API! By using this C-style interface, we got the benefits of ALP's high-performance GraphBLAS engine without having to dive into template programming or parallelization details. From here, you can explore other parts of ALP's API: for instance, ALP/GraphBLAS provides a full GraphBLAS-compatible API for linear algebra operations on vectors and matrices, and ALP also offers a Pregel-like graph processing interface. All of these can be integrated with existing code in a similar fashion (just link against ALP's libraries) [2][3].

In summary, ALP's transition paths enable a smooth adoption of advanced HPC algorithms – you write standard code, and ALP handles the rest. We focused on the CG solver, but the same idea applies to other supported interfaces. Feel free to experiment with different matrices, add a custom preconditioner function to see its effect, or try other solvers if ALP introduces them in future releases. Happy coding!

References

[1] ALP v0.8 Transition Path Overview.
`http://albert-jan.yzelman.net/alp/v0.8-preview/group__TRANS.html`

[2] ALP Documentation and Examples. GitHub.
`https://github.com/Algebraic-Programming/ALP`

[3] Use ALP GraphBLAS in Your Project. GitHub Docs.
`https://github.com/Algebraic-Programming/ALP/blob/develop/docs/Use_ALPGraphBLAS_in_your_own_project.md`

[4] Using ALP Transition Interfaces. GitHub Docs.
`https://github.com/Algebraic-Programming/ALP/blob/develop/docs/Transition_use.md`

This document was generated on July 23, 2025.