RC25508 (WAT1411-052) November 19, 2014
Computer Science

IBM Research Report

Graph Programming Interface: Rationale and Specification

K. Ekanadham, Bill Horn, Joefon Jann, Manoj Kumar, José Moreira, Pratap Pattnaik, Mauricio Serrano, Gabi Tanase, Hao Yu
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

Research Division
Almaden – Austin – Beijing – Cambridge – Dublin - Haifa – India – Melbourne - T.J. Watson – Tokyo - Zurich

Graph Programming Interface: Rationale and Specification

K Ekanadham, Bill Horn, Joefon Jann, Manoj Kumar,
José Moreira, Pratap Pattnaik, Mauricio Serrano, Gabi Tanase, Hao Yu

Generated on 2014/11/12 at 06:28:15 EDT

Abstract

Graph Programming Interface (GPI) is an interface for writing graph algorithms using linear algebra formulation. The interface addresses the requirements of supporting both portability and high performance across a wide spectrum of computing platforms. The application developer composes his or her application using a collection of objects and methods defined by GPI. This application is then linked with a run-time library that implements the objects and methods efficiently on the target platform. This run-time library can be optimized to use the memory hierarchy characteristics and parallelism features of that platform. Concrete instances of GPI must follow a specific binding of the interface to a programming language. We first present a binding for the C programming language.

Contents

1 Introduction

Graph Programming Interface (GPI) is an interface for writing graph algorithms using linear algebra formulation [1]. The algorithms are independent of both the programming language and the implementation of data structures defined in the interface. GPI defines a set of objects and the semantics of methods that operate on those objects. In order to implement the objects and methods of GPI in a specific language, one needs to specify GPI *binding* rules specific to that language. In the appendix, we present the bindings for the C programming language. In the future, bindings for other languages can be suitably defined.

We start this report with a motivation for defining GPI: the desire to simultaneously achieve high performance and high portability for graph analytics applications. We proceed with an example of a usage scenario for GPI-based code in a larger graph analytics context. We then present the definition of the GPI interface in a programming language-independent form. We conclude with a summary of observations about GPI. The appendix provides a binding of GPI for the C programming language (C11 standard), various examples of graph algorithms coded in C using GPI, and an overview of random graph generation techniques that are useful for the study of graph analytics algorithms.

2 Motivation

Graph analytics is an important component of modern business computing. Much of the big data information, a subject commanding great attention these days, is graph structured. Graph analytics techniques require these large graphs to be sub-graphed (analogous to select and project operations in relational databases) and be analyzed for various properties.

2.1 The limitations of current graph analytics approaches

Since large graph analysis requires sizable computational resources, application developers are often faced with the task of selecting the hardware system to execute the applications. The potential systems one considers are large SMPs (*i.e.*, multi-core/multi-threaded general purpose processors based systems), distributed memory systems, and accelerated systems where the CPUs are augmented with GPUs and/or FPGAs. Traditionally, to achieve good performance, each of these systems requires the basic kernels of the graph algorithm to be specialized for that system.

The application programmer is then faced with the challenge of achieving high performance while maintaining code portability, either because systems evolve over time or because the user wishes to move from one type of system to another. Attaining optimum performance requires detailed knowledge of the design of the processors and the systems they comprise, including their cache and memory hierarchy. This knowledge is needed to adapt the analytics algorithms to the underlying system, in particular to take advantage of the parallelism or concurrency at the chip, node or system level. Exploiting concurrency and parallelism so far has been and still is a skilled and non-automated task.

The performance and portability challenge is not easily addressable by compilers because compilers do not have the ability to examine large instruction windows. Furthermore, in the conventional representation of graph algorithms the control flow is often data dependent, making the job of the compiler even more difficult. Particularly, the number of vertices and edges in the graph being analyzed, the sparsity of the graph and whether that sparsity have a regular pattern, are not know to the compiler.

2.2 The value of a graph programming interface

Our approach is to define a programming interface that can be used to build machine-independent graph algorithms. This programming interface can be implemented through a platform-optimized run-time library that is linked with the application code. Such approach bypasses the compiler difficulties mentioned above and is designed to provide the following values and capabilities:

1. Unburden the application developer from performance concerns at the processor level.
2. Unburden the application developer from the task of adapting his or her code to the sizes of the various caches and other characteristics of the memory hierarchy.
3. Unburden the application developer from the headaches of exploiting parallelism, factoring in the design of the memory subsystem as well as the nature of compute nodes.
4. As a consequence of the above, address the porting issue while delivering high performance.

The interface described in this paper creates the above values by providing a set of well defined objects and methods on those objects that support a linear algebra formulation of graph algorithms. That is, the algorithms are expressed as a combination of operations on vectors and matrices. In particular, operations on the adjacency matrix of a graph. By optimizing the methods of the interface for a platform, we can ensure that the algorithms so described will run well in that platform. Furthermore, the algorithms are portable (and performance portable) across a spectrum of platforms that efficiently implement these methods.

Ideally, a run-time library implementing the GPI interface will use two run-time inputs to produce an optimized execution of the graph algorithm, as shown in Figure 1. The first input is information on the characteristics of the graph, including features such as the size of the graph and nature of its sparsity. The second input is information on attributes of the execution platform, such as the size of the main memory and cache hierarchies, SMT levels of cores and performance projections for such levels.

Our goal is to define GPI so that it can be implemented efficiently across a wide spectrum of platforms, and thus deliver on the *performance with portability* value proposition. We also want to develop initial reference implementations that demonstrate that value across selected platforms. Over time, the specification of GPI will evolve as new systems appear and new algorithms are covered. The implementations will also get better as more effort and experience is thrown into the problem.

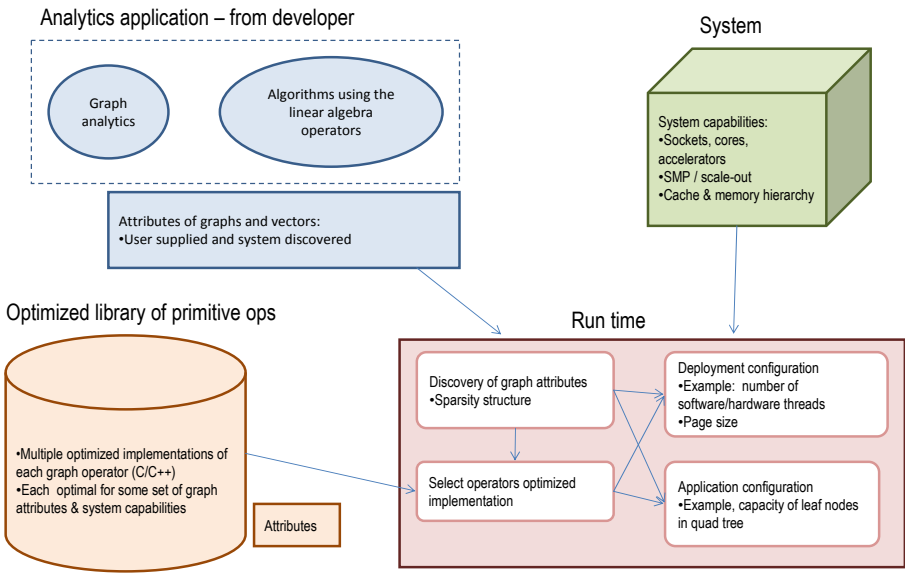


Figure 1: High level view of a graph analytics run-time.

2.3 The linear algebra approach

We chose to define GPI according to a linear algebra formulation of graph algorithms [1]. This formulation centers around operations on the adjacency matrix A of a graph. For a graph of n vertices, this is an $n \times n$ matrix where the elements represent the edges of the graph. For unweighted graph, A is a matrix of Boolean values: $A_{ij} = \mathsf{true}$ means that there is an edge from vertex i to vertex j. Correspondingly, $A_{ij} = \mathsf{false}$ means no such edge. For weighted graphs, A_{ij} takes numerical values. $A_{ij} = 0$ means no edge from vertex i to vertex j. Correspondingly, a positive value means an edge of that weight from vertex i to vertex j. For undirected graphs, the adjacency matrix is always symmetric.

In the linear algebra formulation, it is also common to represent sets of vertices by vectors. If there are n vertices in the graph, then a vector v of size n is used, such that $v_i = 0$ (or false) indicate vertex i does not belong to the set. Correspondingly, $v_i > 0$ (or true) indicates vertex i does belong to the set. Vectors can also be used to represent vertex labels, number of paths to a vertex, reachability sets and other quantities.

The linear algebra formulation for graph algorithms has several value propositions. First, by reducing the computation to linear algebra operations, we naturally get a framework to exploit parallelism and memory hierarchies of various computing platforms. The formulation also delivers the sought off portability, since a library of linear algebra operations can be optimized for a specific platform while preserving a standard interface. Productivity is also enhanced through the definition of a well formalized set of building blocks that can be used to encode general algorithms. Further-

more, these building blocks are already familiar to people from the more traditional scientific and technical computing domains. Finally, the linear algebra formulation supports a perturbative approach to computation. If a computation C is performed in an adjacency matrix A producing a result $y = C(A)$, then the effect on the result of a small change δA to the adjacency matrix can often be approximated (if not computed exactly) as $\delta y = C'(\delta A)$ where C' is related to C (sometimes identical). As a simple example, consider the matrix-vector product: $(A + \delta A)x = Ax + \delta Ax$.

3 GPI scope: a usage scenario

Applications such as graph database, graph analysis and social network analysis software handle a large number of entities and a variety of relationships among those entities. The graphs underlining these applications typically carry a large number of attributes on the vertices and edges. These graphs are persistently stored in files or database management systems. Examples include Neo4J, Accumulo, and HBASE. The graph data may also be persisted using formats such as CSV, GraphML, Graph eXchange Language (GXL), and Resource Description Framework (RDF).

The design point for GPI is focused on the structural aspects of the graph representations. More specifically, using the adjacency matrix of the graph. Figure 2 illustrates a typical usage scenario where GPI will be used in today's graph analytics applications. Starting with the persistent graph data, applications often perform filtering of a graph to create a subgraph. From that subgraph, the application extracts its structure, using various extractor functions, in the form of its adjacency matrix. That structure is then manipulated using GPI objects and methods. The extractor modules provide functionality to manage and populate the adjacency matrix, leaving the high level graph data parsing and the extraction of other graph data to other components.

To exercise the separation of GPI interface and implementation, we keep the definition of the adjacency matrix, as well as other matrices and vectors, encapsulated from the users. These matrices and vectors can only be manipulated by GPI methods. That way, data representation for objects and algorithms for methods can be chosen as to optimize for a particular platform and data set combination. In the future, GPI will define a service interface to provide utilities for graph dumping, debugging, and monitoring.

4 The interface

This section of the report defines the interface that any GPI implementation must conform to. The interface specification consists of two parts: First, we introduce the *objects* defined by GPI. GPI defines *scalar*, *vector* and *matrix* objects as containers of data. GPI also defines function objects, that perform transformations on the data in these containers. Second, GPI defines a set of methods to operate on its objects, implementing linear algebra operations on matrices and vectors. GPI objects are fully opaque, in that they can only be operated on by the GPI methods. We proceed to explain each of these parts of GPI.

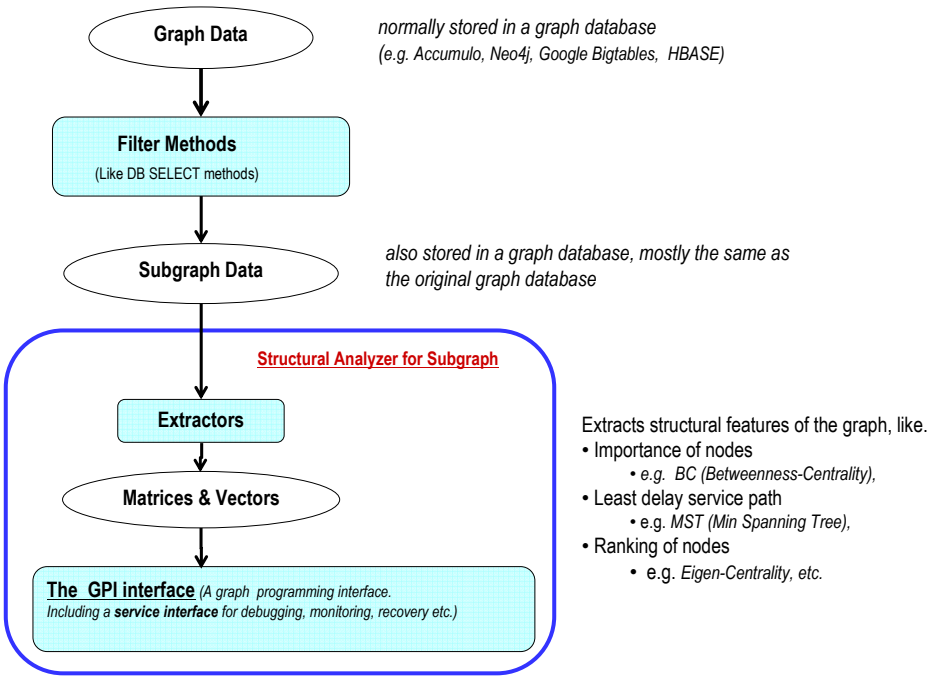


Figure 2: Usage scenario for GPI in graph analytics work flow.

4.1 Objects

A summary of GPI objects is shown in Figure 3. GPI_Scalar, GPI_Vector and GPI_Matrix are all containers of homogeneous elements. Each element is one data item, and the type of the container is the type of its elements. Supported element types are shown in Table 1. We use GPI_type to denote one of the element data types in the table. GPI_Function is the fourth type of object in GPI. All these objects are discussed in more detail below.

When a value of type T_1 is assigned to an element (from a scalar, vector or matrix) of type T_2, *casting* has to occur. Casting is the transformation of a value of one type to a value of another type. Some casting operations are lossless, in the sense that no information is lost. Other cast operations are lossy, because information in the original type cannot be preserved in the new type. For example, casting from GPI_int32 to GPI_fp64 is always lossless. Every 32-bit signed integer has an exact representation as a 64-bit floating-point number. The reverse casting is lossy, as most 64-bit floating-point numbers do not have an exact representation as a 32-bit integer. Casting is performed automatically by GPI inside its methods, whenever a computed value is of a type different than the container it is supposed to be stored in. Application programmer code can also perform casting explicitly. This can be done by using one of the copy methods described below.

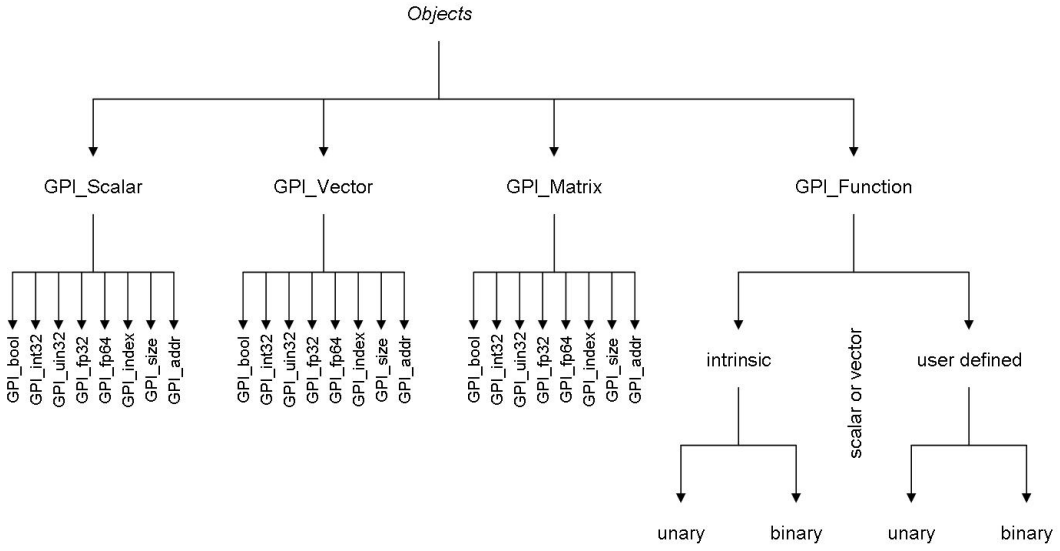


Figure 3: Objects in GPI.

4.1.1 GPI_Scalar

GPI_Scalar is a zero-dimensional, one-element container. The element can be of any of the types in Table 1. GPI defines a number of scalars, shown in Table 2. Other scalars can be built by casting elements of one of the types in Table 1 to GPI_Scalar. (The casting mechanism is specific to each programming language binding. We discuss casting for the C programming language in the appendix.)

Table 1: Data types in GPI.

GPI data type	Description
GPI_bool	Boolean (true or false)
GPI_int32	32-bit signed integer
GPI_uint32	32-bit unsigned integer
GPI_fp32	32-bit IEEE floating-point number
GPI_fp64	64-bit IEEE floating-point number
GPI_index	A matrix (row or column) or vector index, a vertex index
GPI_size	The size (number of elements) of a collection
GPI_addr	An address (pointer)

Table 2: Built-in GPI scalars.

Built-in scalar	value
GPI_zero	0
GPI_one	1
GPI_minusone	-1
GPI_null	nothing

4.1.2 GPI_Vector

A *vector* is a one-dimensional homogeneous container defined by the type T of its elements and the number n of elements in the container (the size of the vector). For a vector v of size n, $v[i]$ denotes the i-the element of that vector, $i = 0, \ldots, n-1$.

Vectors in GPI are represented by variables of type GPI_Vector. Let v be a vector of type T. We say that a GPI_Vector v is *associated* with vector v if and only if $\mathsf{v}[i] \equiv v[i], \forall i$. Reading or writing an element $\mathsf{v}[i]$ has the effect of reading or writing the corresponding associated element $v[i]$. More than one GPI_Vector can be associated with the same vector.

4.1.3 GPI_Matrix

A *matrix* is a two-dimensional homogeneous container defined by the type T of its elements, the number m of rows and the number n of columns in the container (the shape of the matrix). For a matrix A of shape $m \times n$, $A[i, j]$ denotes the element at row i and column j of the matrix, $i = 0, \ldots, m-1$, $j = 0, \ldots, n-1$. $A[i, :]$ denotes the i-th row of the matrix (an n-element vector), and $A[:, j]$ denotes the j-th column of the matrix (an m-element vector).

Matrices in GPI are represented by variables of type GPI_Matrix. Let A be a matrix of type T. We say that a GPI_Matrix A is *associated* with matrix A if and only if $\mathsf{A}[i, j] \equiv A[i, j], \forall i, j$. Reading or writing an element $\mathsf{A}[i, j]$ has the effect of reading or writing the corresponding associated element $A[i, j]$. More than one GPI_Matrix can be associated with the same matrix. Furthermore, a GPI_Vector can be associated with a row or column of a matrix.

4.1.4 GPI_Function

GPI supports both unary (one input argument) and binary (two input arguments) functions. Furthermore, functions can be either *scalar* (taking scalar arguments) or *vector* (taking vector arguments) functions. A unary scalar function f must have the signature $f(\mathsf{GPI\_Scalar}) \rightarrow \mathsf{GPI\_Scalar}$. A binary scalar function g must have the signature $g(\mathsf{GPI\_Scalar} \times \mathsf{GPI\_Scalar}) \rightarrow \mathsf{GPI\_Scalar}$. Correspondingly, a unary vector function f must have the signature $f(\mathsf{GPI\_Vector}) \rightarrow \mathsf{GPI\_Vector}$, whereas a binary vector function g must have the signature $g(\mathsf{GPI\_Vector} \times \mathsf{GPI\_Vector}) \rightarrow \mathsf{GPI\_Vector}$. The type of a function is defined by the types of its input and output arguments. GPI functions can be either intrinsic (built-in) to the interface definition or user defined. The set of GPI scalar built-in functions, is listed in Table 3. When used as vector functions, these built-ins apply element-wise.

Function GPI_LOC is a bit of a special case. It is used where a binary function is expected (e.g., reduction operations on vectors, see GPI_reduce below). It returns the location in the vector of the first occurrence of the starting value.

Table 3: Built-in functions in GPI.

Function	meaning
GPI_MAX	maximum
GPI_MIN	minimum
GPI_SUM	sum
GPI_PROD	product
GPI_LAND	logical and
GPI_BAND	bit-wise and
GPI_LOR	logical or
GPI_BOR	bit-wise or
GPI_LXOR	logical xor
GPI_BXOR	bit-wise xor
GPI_EQ	equal
GPI_NE	not equal
GPI_GT	greater than
GPI_GE	greater than or equal
GPI_LT	less than
GPI_LE	less than or equal
GPI_LOC	location of a value

(a) binary functions

Function	meaning
GPI_NEG	negation
GPI_LNOT	logical not
GPI_BNOT	bit-wise not

(b) unary functions

4.2 Methods

GPI defines four groups of methods. *Vector and matrix* building methods (also called *entity* methods) are used to create and destroy vectors and matrices. *Accessor* methods are used to manipulate the internals of vectors and matrices. *Base* methods are the building blocks of graph algorithms in linear algebra formulation. Finally, *derived* methods can be constructed from the base methods but can be implemented more efficiently directly and are thus part of the GPI interface.

Several methods accept an optional *mask* parameter. The mask is a vector of elements that controls conditional execution inside the method. Element values are interpreted as either false (a value of 0) or true (any value not equal to 0). Whenever a mask parameter is not specified when calling the method, this is equivalent to passing a mask with all elements true.

4.2.1 Vector and matrix building methods

GPI_Vector_new(v,type,size)

v	GPI_Vector	OUT	a new vector
type	GPI_type	IN	type of the vector elements
size	GPI_size	IN	number of elements in vector

Creates a new vector v with size elements of type type and associates it with v. The value of each element is the corresponding zero value for that type.

GPI_Vector_copy(v,u[,m])

v	GPI_Vector	OUT	output vector
u	GPI_Vector	IN	input vector
m	GPI_Vector	IN	optional mask

Let v be a vector of type T_1 and size n and let u be a vector of type T_2 and size n. This method copies the value of each element of u into the corresponding element of v, casting the value as necessary ($\mathsf{v}[i] \leftarrow (T_1)\mathsf{u}[i], \forall i \mid \mathsf{m}[i] = \mathsf{true}$).

GPI_Vector_delete(v)

v	GPI_Vector	IN	A vector

Destroys the vector associated with v. The GPI_Vector_delete method must be called at least once for each vector created with GPI_Vector_new. The interval between the creation and destruction of a vector is its *lifetime*. Once a vector is destroyed, it should not be subsequently referenced by any variable that was associated with it.

GPI_Matrix_new(A,type,rows,cols)

A	GPI_Matrix	OUT	new matrix
type	GPI_type	IN	type of the matrix elements
rows	GPI_size	IN	number of rows in matrix
cols	GPI_size	IN	number of columns in matrix

Creates a new matrix A with elements of type type and shape rows $\times$ cols, and associates it with A. The value of each element is the corresponding zero value for that type.

GPI_Matrix_copy(B,A)

B	GPI_Matrix	OUT	output matrix
A	GPI_Matrix	IN	input matrix

Let B be a matrix of type T_1 and shape $m \times n$. Let A be a matrix of type T_2 and shape $m \times n$. This method copies the values of each element of A into the corresponding element of B, performing casting as necessary ($\mathsf{B}[i,j] \leftarrow (T_1)\mathsf{A}[i,j], \forall i,j$).

GPI_Matrix_copy(B,A,dim[,m])

B	GPI_Matrix	OUT	output matrix
A	GPI_Matrix	IN	input matrix
dim	GPI_index	IN	dimension
m	GPI_Vector	IN	optional mask

Let B be a matrix of type T_1 and shape $m \times n$. Let A be a matrix of type T_2 and shape $m \times n$. If ($\mathsf{dim} = 0$), this method copies each row of A into the corresponding row of B ($\mathsf{B}[i,:] \leftarrow \mathsf{A}[i,:], \forall i \mid$

m$[i]$ = true). If (dim = 1), this method copies each column of A into the corresponding column of B (B$[:, j] \leftarrow$ A$[:, j], \forall j \mid$ m$[j]$ = true).

GPI_Matrix_delete(A)

A	GPI_Matrix	IN	matrix

Destroys the matrix associated with A. The GPI_Matrix_delete method must be called at least once for each matrix created with GPI_Matrix_new. The interval between the creation and destruction of a matrix is its *lifetime*.

4.2.2 Accessor methods

GPI_Matrix_nrows(nrows,A)

nrows	GPI_size	OUT	number of rows in matrix
A	GPI_Matrix	IN	matrix

Returns in nrows the number of rows of the matrix associated with matrix.

GPI_Matrix_ncols(ncols,A)

ncols	GPI_size	OUT	number of columns in matrix
A	GPI_Matrix	IN	matrix

Returns in ncols the number of columns of the matrix associated with matrix.

GPI_Matrix_row(row,A,index)

row	GPI_Vector	OUT	vector
A	GPI_Matrix	IN	matrix
index	GPI_index	IN	the index of a row of the matrix

Associates row with the vector A[index, :].

GPI_Matrix_col(col,A,index)

col	GPI_Vector	OUT	vector
A	GPI_Matrix	IN	matrix
index	GPI_index	IN	the index of a column of the matrix

Associates col with the vector A[:, index].

GPI_Matrix_getElement(element,A,row,col)

element	GPI_Scalar	OUT	element
A	GPI_Matrix	IN	matrix
row	GPI_index	IN	the index of a row of the matrix
col	GPI_index	IN	the index of a column of the matrix

Returns in element the value of A[row, col].

GPI_Matrix_setElement(A,row,col,element)

A	GPI_Matrix	IN	matrix
row	GPI_index	IN	the index of a row of the matrix
col	GPI_index	IN	the index of a column of the matrix
element	GPI_Scalar	IN	element

Sets the value of A[row, col] to the value of element, casting if necessary.

GPI_Vector_size(size,v)

size	GPI_size	OUT	number of elements
v	GPI_Vector	IN	vector

Returns in size the number of elements of v.

GPI_Vector_getElement(element,v,index)

element	GPI_Scalar	OUT	element
v	GPI_Vector	IN	vector
index	GPI_index	IN	the index of an element of the vector

Returns in element the value of v[index].

GPI_Vector_setElement(v,index,element)

v	GPI_Vector	IN	vector
index	GPI_index	IN	the index of an element of the vector
element	GPI_Scalar	IN	element

Sets the value of v[index] to the value of element, casting if necessary.

4.2.3 Base methods

Several base methods in GPI are *polymorphic*. They apply to different combinations of input types, typically vectors and matrices. In the following descriptions, methods are grouped by the particular kind of operation they perform. Within each group, we list the different forms of the methods.

4.2.3.1 Replication

GPI_replicate(v,x[,m])

v	GPI_Vector	OUT	output vector
x	GPI_Scalar	IN	input scalar
m	GPI_Vector	IN	optional mask

Let x be a scalar of type T and v be a vector of type T and size n. This method set each element of v to the value of x ($\mathsf{v}[i] \leftarrow \mathsf{x}, \forall i \mid \mathsf{m}[i] = \mathsf{true}$).

GPI_replicate(A,x)

A	GPI_Matrix	OUT	output matrix
x	GPI_Scalar	IN	input scalar

Let x be a scalar of type T and A be a matrix of type T and shape $m \times n$. This method sets each element of A to the value of x ($\mathsf{A}[i,j] \leftarrow \mathsf{x}, \forall i,j$).

GPI_replicate(A,x,dim[,m])

A	GPI_Matrix	OUT	output matrix
x	GPI_Vector	IN	input vector
dim	GPI_index	IN	replicating dimension
m	GPI_Vector	IN	optional mask

Let x be a vector of type T and size p and A be a matrix of type T and shape $m \times n$. If ($\mathsf{dim} = 0$), This method sets each row of A to the value of x ($\mathsf{A}[i,:] \leftarrow \mathsf{x}, \forall i \mid \mathsf{m}[i] = \mathsf{true}, p = n$). If ($\mathsf{dim} = 1$), This method sets each column of A to the value of x ($\mathsf{A}[:,j] \leftarrow \mathsf{x}, \forall j \mid \mathsf{m}[j] = \mathsf{true}, p = m$).

4.2.3.2 Index generation

GPI_indices(v[,m])

v	GPI_Vector	OUT	output vector
m	GPI_Vector	IN	optional mask

Let v be a vector of size n. This function sets the value of each element of v to its index ($\mathsf{v}[i] \leftarrow i, \forall i \mid \mathsf{m}[i] = \mathsf{true}$).

4.2.3.3 Filtering

GPI_filter(w,x,u,v[,m])

w	GPI_Vector	OUT	output vector
x	GPI_Scalar	IN	test value
u	GPI_Vector	IN	input vector
v	GPI_Vector	IN	input vector
m	GPI_Vector	IN	optional mask

Given three vectors, w, u and v of type T and size n, and a scalar x of type T, it computes vector w such that $\mathsf{w}[i] \leftarrow ((\mathsf{u}[i] = x)?\mathsf{v}[i] : x), \forall i \mid \mathsf{m}[i] = \mathsf{true}$.

GPI_filter(C,x,A,B,dim[,m])

C	GPI_Matrix	OUT	output matrix
x	GPI_Vector	IN	test values
A	GPI_Matrix	IN	input matrix
B	GPI_Matrix	IN	input matrix
dim	GPI_index	IN	filtering dimension
m	GPI_Vector	IN	optional mask

If $\mathsf{dim} = 0$, this is equivalent to $\mathsf{GPI\_filter}(\mathsf{C}[i,:], \mathsf{x}[i], \mathsf{A}[i,:], \mathsf{B}[i,:][,m]), \forall i$. If $\mathsf{dim} = 1$, this is equivalent to $\mathsf{GPI\_filter}(\mathsf{C}[:,j], \mathsf{x}[j], \mathsf{A}[:,j], \mathsf{B}[:,j][,m]), \forall j$.

GPI_filterneg(w,x,u,v[,m])

w	GPI_Vector	OUT	output vector
x	GPI_Scalar	IN	test value
u	GPI_Vector	IN	input vector
v	GPI_Vector	IN	input vector
m	GPI_Vector	IN	optional mask

Given three vectors, w, u and v of type T and size n, and a scalar x of type T, it computes vector w such that $\mathsf{w}[i] \leftarrow ((\mathsf{u}[i] \neq x)?\mathsf{v}[i] : x), \forall i \mid \mathsf{m}[i] = \mathsf{true}$.

GPI_filterneg(C,x,A,B,dim[,m])

C	GPI_Matrix	OUT	output matrix
x	GPI_Vector	IN	test values
A	GPI_Matrix	IN	input matrix
B	GPI_Matrix	IN	input matrix
dim	GPI_index	IN	filtering dimension
m	GPI_Vector	IN	optional mask

If $\mathsf{dim} = 0$, this is equivalent to $\mathsf{GPI\_filterneg}(\mathsf{C}[i,:], \mathsf{x}[i], \mathsf{A}[i,:], \mathsf{B}[i,:][,m]), \forall i$. If $\mathsf{dim} = 1$, this is equivalent to $\mathsf{GPI\_filterneg}(\mathsf{C}[:,j], \mathsf{x}[j], \mathsf{A}[:,j], \mathsf{B}[:,j][,m]), \forall j$.

4.2.3.4 Reduction

GPI_reduce(y,f,x,u[,m])

y	GPI_Scalar	OUT	output scalar
f	GPI_Function	IN	function
x	GPI_Scalar	IN	starting value
u	GPI_Vector	IN	input vector
m	GPI_Vector	IN	optional mask

Given a scalar x of type T_2, a vector u of type T_1 and size n, and a scalar function $\mathsf{f}(T_1, T_2) \rightarrow T_2$, compute a scalar y of type T_2 using the recurrence $\mathsf{y} \leftarrow \mathsf{x}$; $\mathsf{y} \leftarrow \mathsf{f}(\mathsf{u}[i], \mathsf{y}), i = 0, \ldots, n-1 \mid \mathsf{m}[i] = \mathsf{true}$.

GPI_reduce(y,f,x,A,dim[,m])

y	GPI_Vector	OUT	output vector
f	GPI_Function	IN	function
x	GPI_Vector	IN	starting values
A	GPI_Matrix	IN	input matrix
dim	GPI_index	IN	reducing dimension
m	GPI_Vector	IN	optional mask

If $(\mathsf{dim} = 0)$, then given a vector x of type T_2 and size n, a matrix A of type T_1 and shape $m \times n$, and a vector function $\mathsf{f}(T_1, T_2) \rightarrow T_2$, compute a vector y of type T_2 and size n using the recurrence $\mathsf{y} \leftarrow \mathsf{x}$; $\mathsf{y} \leftarrow \mathsf{f}(\mathsf{A}[i,:], \mathsf{y}), i = 0, \ldots, m-1 \mid \mathsf{m}[i] = \mathsf{true}$. If $(\mathsf{dim} = 1)$, then given a vector x of type T_2 and size m, a matrix A of type T_1 and shape $m \times n$, and a vector function $\mathsf{f}(T_1, T_2) \rightarrow T_2$, compute a vector y of type T_2 and size m using the recurrence $\mathsf{y} \leftarrow \mathsf{x}$; $\mathsf{y} \leftarrow \mathsf{f}(\mathsf{A}[:,j], \mathsf{y}), j = 0. \ldots, n-1 \mid \mathsf{m}[j] = \mathsf{true}$.

4.2.3.5 Mapping

GPI_map(v,f,u[,m])

v	GPI_Vector	OUT	output vector
f	GPI_Function	IN	function
u	GPI_Vector	IN	input vector
m	GPI_Vector	IN	optional mask

Given a scalar function $\mathsf{f}(T_1) \to T_2$, a vector u of type T_1 and size n, and a vector v of size n and type T_2, it computes $\mathsf{v}[i] \leftarrow \mathsf{f}(\mathsf{u}[i]), \forall i \mid \mathsf{m}[i] = \mathsf{true}$.

GPI_map(B,f,A,dim[,m])

B	GPI_Matrix	OUT	output matrix
f	GPI_Function	IN	function
A	GPI_Matrix	IN	input matrix
dim	GPI_index	IN	mapping dimension
m	GPI_Vector	IN	optional mask

If $(\mathsf{dim} = 0)$, then given a vector function $\mathsf{f}(T_1) \to T_2$, a matrix A of type T_1 and shape $m \times n$, and a matrix B of shape $m \times n$ and type T_2, it computes $\mathsf{B}[i,:] \leftarrow \mathsf{f}(\mathsf{A}[i,:]), \forall i \mid \mathsf{m}[i] = \mathsf{true}$. If $(\mathsf{dim} = 1)$, then given a vector function $\mathsf{f}(T_1) \to T_2$, a matrix A of type T_1 and shape $m \times n$, and a matrix B of shape $m \times n$ and type T_2, it computes $\mathsf{B}[:,j] \leftarrow \mathsf{f}(\mathsf{A}[:,j]), \forall j \mid \mathsf{m}[j] = \mathsf{true}$.

4.2.3.6 Zipping

GPI_zip(w,f,u,v[,m])

w	GPI_Vector	OUT	output vector
f	GPI_Function	IN	function
u	GPI_Vector	IN	input vector
v	GPI_Vector	IN	input vector
m	GPI_Vector	IN	optional mask

Given a scalar function $\mathsf{f}(T_1, T_2) \to T_3$, a vector u of type T_1 and size n, a vector v of type T_2 and size n, and a vector w of size n and type T_3, it computes $\mathsf{w}[i] \leftarrow \mathsf{f}(\mathsf{u}[i], \mathsf{v}[i]), \forall i \mid \mathsf{m}[i] = \mathsf{true}$.

GPI_zip(C,f,A,B,dim[,m])

C	GPI_Matrix	OUT	output matrix
f	GPI_Function	IN	function
A	GPI_Matrix	IN	input matrix
B	GPI_Matrix	IN	input matrix
dim	GPI_index	IN	zipping dimension
m	GPI_Vector	IN	optional mask

If $(\mathsf{dim} = 0)$, then given a vector function $\mathsf{f}(T_1, T_2) \to T_3$, a matrix A of type T_1 and shape $m \times n$, a matrix B of shape $m \times n$ and type T_2, and a matrix C of shape $m \times n$ and type T_3 it computes $\mathsf{C}[i,:] \leftarrow \mathsf{f}(\mathsf{A}[i,:], \mathsf{B}[i,:]), \forall i \mid \mathsf{m}[i] = \mathsf{true}$. If $(\mathsf{dim} = 1)$, then given a vector function $\mathsf{f}(T_1, T_2) \to T_3$,

a matrix A of type T_1 and shape $m \times n$, a matrix B of shape $m \times n$ and type T_2, and a matrix C of shape $m \times n$ and type T_3 it computes $\mathsf{C}[:, j] \leftarrow \mathsf{f}(\mathsf{A}[:, j], \mathsf{B}[:, j]), \forall j \mid \mathsf{m}[j] = \mathsf{true}$.

4.2.3.7 Function application

GPI_apply(v,f,u,k[,m])

v	GPI_Vector	OUT	output vector
f	GPI_Function	IN	function
u	GPI_Vector	IN	initial values
k	GPI_uint32	IN	number of applications
m	GPI_Vector	IN	optional mask

Given a vector function $\mathsf{f}(T) \rightarrow T$, an initial value vector u of type T and size n, and a non-negative integer k, it computes an output vector v of size n and type T through $((\mathsf{k} = 0)?\mathsf{v}[i] \leftarrow \mathsf{u}[i] \mid \mathsf{m}[i] = \mathsf{true} : \mathsf{apply}(\mathsf{v}, \mathsf{f}, \mathsf{f}(\mathsf{u}), \mathsf{k} - 1))$.

GPI_apply(B,f,A,k,dim[,m])

B	GPI_Matrix	OUT	output matrix
f	GPI_Function	IN	function
A	GPI_Matrix	IN	initial values
k	GPI_uint32	IN	number of applications
dim	GPI_index	IN	applying dimension
m	GPI_Vector	IN	optional mask

If $(\mathsf{dim} = 0)$, then this computes $\mathsf{GPI\_apply}(\mathsf{B}[i, :], \mathsf{f}, \mathsf{A}[i, :], \mathsf{k}), \forall i \mid \mathsf{m}[i] = \mathsf{true}$. If $(\mathsf{dim} = 1)$, then this computes $\mathsf{GPI\_apply}(\mathsf{B}[:, j], \mathsf{f}, \mathsf{A}[:, j], \mathsf{k}), \forall j \mid \mathsf{m}[j] = \mathsf{true}$.

GPI_fixpt(v,f,u[,m])

v	GPI_Vector	OUT	output vector
f	GPI_Function	IN	function
u	GPI_Vector	IN	initial values
m	GPI_Vector	IN	optional mask

Given a vector function $\mathsf{f}(T) \rightarrow T$, and an initial value vector u of type T and size n, it computes $\mathsf{apply}(\mathsf{v}, \mathsf{f}, \mathsf{u}, k[, \mathsf{m}])$, where k is the smallest non-negative value such that $\mathsf{apply}(\mathsf{v}, \mathsf{f}, \mathsf{u}, k[, \mathsf{m}])$ and $\mathsf{apply}(\mathsf{v}, \mathsf{f}, \mathsf{u}, k + 1[, \mathsf{m}])$ return the same $\mathsf{v}[i], \forall i \mid \mathsf{m}[i] = \mathsf{true}$. The value of v is undefined if there is no such k.

GPI_fixpt(B,f,A,dim[,m])

B	GPI_Matrix	OUT	output matrix
f	GPI_Function	IN	function
A	GPI_Matrix	IN	initial values
dim	GPI_index	IN	applying dimension
m	GPI_Vector	IN	optional mask

If $(\mathsf{dim} = 0)$, then this computes $\mathsf{GPI\_fixpt}(\mathsf{B}[i, :], \mathsf{f}, \mathsf{A}[i, :]), \forall i \mid \mathsf{m}[i] = \mathsf{true}$. If $(\mathsf{dim} = 1)$, then

this computes GPI_fixpt(B[:, j], f, A[:, j]), $\forall j \mid$ m[j] = true.

4.2.3.8 Transposition

GPI_transpose(B,A)

B	GPI_Matrix	OUT	output matrix
A	GPI_Matrix	IN	input matrix

Let A be a matrix of type T and shape $m \times n$. This function associates with B the transpose of A. That is, B[i, j] $\equiv$ A[j, i], $\forall i, j$. Modifying B[i, j] has the effect of modifying A[j, i] and vice versa.

4.2.4 Derived methods

Derived methods can also be polymorphic. We again group the methods by kind of operations and within each group we list the different forms of the methods.

4.2.4.1 Generalized inner product

GPI_innerp(y,f,g,x,u,v[,m])

y	GPI_Scalar	OUT	result scalar
f	GPI_Function	IN	reduce function
g	GPI_Function	IN	map function
x	GPI_Scalar	IN	initial scalar
u	GPI_Vector	IN	input vector
v	GPI_Vector	IN	input vector
m	GPI_Vector	IN	optional mask

Let u and v be vector of size n and types T_1 and T_2 respectively. Let f($T_3 \times T_4$) $\rightarrow T_4$ and g($T_1 \times T_2$) $\rightarrow T_3$ be functions, and let y be of type T_4. This method first computes a vector w with GPI_zip(w, g, u, v[, m]). Then, it computes and returns y using GPI_reduce(y, f, x, w[, m]).

GPI_innerp(y,f,g,x,A,B,dim[,m])

y	GPI_Vector	OUT	result vector
f	GPI_Function	IN	reduce function
g	GPI_Function	IN	map function
x	GPI_Vector	IN	initial vector
A	GPI_Matrix	IN	input matrix
B	GPI_Matrix	IN	input matrix
dim	GPI_index	IN	reducing dimension
m	GPI_Vector	IN	optional mask

Let A and B be matrices of shape $m \times n$ and types T_1 and T_2 respectively. Let f($T_3 \times T_4$) $\rightarrow T_4$ and g($T_1 \times T_2$) $\rightarrow T_3$ be functions, and let y be of type T_4. This method first computes a matrix

C of type T_3 and shape $m \times n$ with GPI_zip(C, g, A, B, dim[, m]). Then, it computes and returns y using GPI_reduce(y, f, x, C, dim[, m]).

4.2.4.2 Generalized matrix multiplication

GPI_mxv(y,A,x,f,g[,m])

y	GPI_Vector	OUT	result vector
A	GPI_Matrix	IN	input matrix
x	GPI_Vector	IN	input vector
f	GPI_Function	IN	reduce function
g	GPI_Function	IN	map function
m	GPI_Vector	IN	optional mask

Given a matrix A of type T_1 and shape $m \times n$, a vector x of type T_2 and size n, functions f$(T_3 \times T_4) \rightarrow T_4$ and g$(T_1 \times T_2) \rightarrow T_3$, and a vector y of type T_4 and size n, this method computes y through GPI_innerp(y$[i]$, f, g, x, A$[i, :]), \forall i \mid$ m$[i]$ = true. If f = GPI_SUM and g = GPI_PROD, this is a standard matrix-vector multiply.

GPI_vxm(y,x,A,f,g[,m])

y	GPI_Vector	OUT	result vector
x	GPI_Vector	IN	input vector
A	GPI_Matrix	IN	A matrix
f	GPI_Function	IN	reduce function
g	GPI_Function	IN	map function
m	GPI_Vector	IN	optional mask

Given a matrix A of type T_2 and shape $m \times n$, a vector x of type T_1 and size m, functions f$(T_3 \times T_4) \rightarrow T_4$ and g$(T_1 \times T_2) \rightarrow T_3$, and a vector y of type T_4 and size n, this method computes y through GPI_innerp(y$[i]$, f, g, x, A$[:, i]), \forall i \mid$ m$[i]$ = true. If f = GPI_SUM and g = GPI_PROD, this is a standard vector-matrix multiply.

GPI_mxm(C,A,B,f,g)

C	GPI_Matrix	OUT	result matrix
A	GPI_Matrix	IN	An input matrix
B	GPI_Matrix	IN	An input matrix
f	GPI_Function	IN	reduce function
g	GPI_Function	IN	map function

Given a matrix A of type T_1 and shape $m \times n$, a matrix B of type T_2 and shape $n \times p$, functions f$(T_3 \times T_4) \rightarrow T_4$ and g$(T_1 \times T_2) \rightarrow T_3$, and a matrix C of type T_4 and shape $m \times p$, this method computes C through GPI_mxv(C$[:, j]$, A, B$[:, j]$, f, g)$, \forall j$. If f = GPI_SUM and g = GPI_PROD, this is a standard matrix-matrix multiply.

4.2.4.3 Graph operations

GPI_successors(y,A,s)

y	GPI_Vector	OUT	result vector
A	GPI_Matrix	IN	adjacency matrix
s	GPI_index	IN	index of a source vertex

Given an adjacency matrix A and a source vertex s for a graph, computes a vector y such that y[i] = true if vertex i can be reached from vertex s in 0 or more steps, and y[i] = false otherwise. We note that y[s] is always true.

GPI_predecessors(y,A,t)

y	GPI_Vector	OUT	result vector
A	GPI_Matrix	IN	adjacency matrix
t	GPI_index	IN	index of a target vertex

Given an adjacency matrix A and a target vertex t for a graph, computes a vector y such that y[i] = true if vertex t can be reached from vertex i in 0 or more steps, and y[i] = false otherwise. We note that y[t] is always true.

GPI_reachability(R,A)

R	GPI_Matrix	OUT	reachability matrix
A	GPI_Matrix	IN	adjacency matrix

Given an adjacency matrix A for a graph, computes the reachability matrix R for that graph. R[i, j] = true if vertex j can be reached from vertex i in 0 or more steps, and R[i, j] = false otherwise. We note that R[i, i] is always true.

5 Conclusions

Graph analytics is an important component of modern business computing. We propose a standard interface called Graph Programming Interface (GPI) to support the development of graph analytics applications that are both portable and high performing. GPI is intended to be used in linear algebra formulation of graph algorithms. For that purpose, it includes objects and methods that implement operations in that domain.

GPI supports scalar, vector and matrix objects. Those objects are completely opaque to the application programmer and can only be manipulated by GPI methods. This approach ensures maximum flexibility for implementations that want to exploit specific machine features and data organization. The same GPI application program can execute efficiently on a variety of machines, including multi-core and many-core processors, GPUs, FPGAs and message-passing systems.

GPI's specification is language independent. A concrete implementation of GPI must conform to a particular programming language binding. In the appendix we present a binding and examples for the C programming language.

References

[1] Jeremy Kepner and John Gilbert. *Graph algorithms in the language of linear algebra*. SIAM 2011, ISBN 978-0-898719-90-1.

[2] Reka Albert and Albert-Laszlo Barbasi. *Topology of Evolving Networks: Local Events and Universality*. Physical Review Letters, Dec. 11 2000, 85(24), pp. 5234-5237.

[3] Deepayan Chakrabarti and Christos Faloutsos. *Graph Mining: Laws, Tools, and Case Studies*. Morgan & Claypool Publishers, ISBN 978-1-608451-15-9, Chapter 11.

A GPI C bindings

In the C language bindings of GPI, every function returns a value of type GPI_int32. When that value is zero (0), the function completed execution without any detected errors. When that return value is any other number, an error was detected during execution of the function. The list of error codes for the C language bindings of GPI is shown in Table 4. The corresponding C types for GPI element types and objects are shown in Table 5.

Table 4: Error codes for the C language bindings of GPI.

error code	error type
< 0	GPI panic (unknown error)
1	inconsistent/invalid parameters
2	out of memory

Table 5: Element data types and objects for the C language bindings of GPI.

GPI name	C name
GPI_bool	GPI_bool
GPI_int32	GPI_int32
GPI_uint32	GPI_uint32
GPI_fp32	GPI_fp32
GPI_fp64	GPI_fp64
GPI_index	GPI_index
GPI_size	GPI_size
GPI_addr	GPI_addr
GPI_Scalar	GPI_Scalar
GPI_Vector	GPI_Vector
GPI_Matrix	GPI_Matrix
GPI_Function	GPI_Function

C language prototypes for the matrix and vector building methods of GPI (so called *entity* methods) are shown in Table 6. C language prototypes for the accessor methods of GPI are shown in Table 7. C language prototypes for the base methods of GPI are shown in Table 8. C language

prototypes for the derived methods of GPI are shown in Table 9. In all cases we omit the return type, which is always `GPI_int32`.

Table 6: Entity methods prototypes for the C language bindings of GPI.

GPI method	C language prototype
GPI_Vector_new	`GPI_Vector_new(GPI_Vector*,GPI_type,GPI_size)`
GPI_Vector_copy	`GPI_Vector_copy(GPI_Vector*,GPI_Vector*,GPI_Vector*)`
GPI_Vector_delete	`GPI_Vector_delete(GPI_Vector*)`
GPI_Matrix_new	`GPI_Matrix_new(GPI_Matrix*,GPI_type,GPI_size,GPI_size)`
GPI_Matrix_copy	`GPI_Matrix_copy(GPI_Matrix*,GPI_Matrix*)`
GPI_Matrix_copy	`GPI_Matrix_copy(GPI_Matrix*,GPI_Matrix*,GPI_index[,GPI_Vector*])`
GPI_Matrix_delete	`GPI_Matrix_delete(GPI_Matrix*)`

Table 7: Accessor methods prototypes for the C language bindings of GPI.

GPI_Matrix_nrows	`GPI_Matrix_nrows(GPI_size*,GPI_Matrix*)`
GPI_Matrix_ncols	`GPI_Matrix_ncols(GPI_size*,GPI_Matrix*)`
GPI_Matrix_row	`GPI_Matrix_row(GPI_Vector*,GPI_Matrix*,GPI_index)`
GPI_Matrix_col	`GPI_Matrix_col(GPI_Vector*,GPI_Matrix*,GPI_index)`
GPI_Matrix_getElement	`GPI_Matrix_getElement(GPI_Scalar*,GPI_Matrix*,GPI_index,GPI_index)`
GPI_Matrix_setElement	`GPI_Matrix_setElement(GPI_Matrix*,GPI_index,GPI_index,GPI_Scalar)`
GPI_Vector_size	`GPI_Vector_size(GPI_size*,GPI_Vector*)`
GPI_Vector_getElement	`GPI_Vector_getElement(GPI_Scalar*,GPI_Vector*,GPI_index)`
GPI_Vector_setElement	`GPI_Vector_setElement(GPI_Vector*,GPI_index,GPI_Scalar)`

Table 8: Base methods prototypes for the C language bindings of GPI.

GPI method	C language prototype
GPI_replicate	GPI_replicate(GPI_Matrix*,GPI_Scalar*)
GPI_replicate	GPI_replicate(GPI_Matrix*,GPI_Vector*,GPI_index[,GPI_Vector*])
GPI_replicate	GPI_replicate(GPI_Vector*,GPI_Scalar*[,GPI_Vector*])
GPI_indices	GPI_indices(GPI_Vector*[,GPI_Vector*])
GPI_filter	GPI_filter(GPI_Vector*,GPI_Scalar*,GPI_Vector*,GPI_Vector*[,GPI_Vector*])
GPI_filter	GPI_filter(GPI_Matrix*,GPI_Vector*,GPI_Matrix*,GPI_Matrix*,GPI_index[,GPI_Vector*])
GPI_filterneg	GPI_filterneg(GPI_Vector*,GPI_Scalar*,GPI_Vector*,GPI_Vector*[,GPI_Vector*])
GPI_filterneg	GPI_filterneg(GPI_Matrix*,GPI_Vector*,GPI_Matrix*,GPI_Matrix*,GPI_index[,GPI_Vector*])
GPI_reduce	GPI_reduce(GPI_Scalar*.GPI_Function*,GPI_Scalar*,GPI_Vector*[,GPI_Vector*])
GPI_reduce	GPI_reduce(GPI_Vector*.GPI_Function*,GPI_Vector*,GPI_Matrix*,GPI_index[,GPI_Vector*])
GPI_map	GPI_map(GPI_Vector*,GPI_Function*,GPI_Vector*[,GPI_Vector*])
GPI_map	GPI_map(GPI_Matrix*,GPI_Function*,GPI_Matrix*,GPI_index[,GPI_Vector*])
GPI_zip	GPI_zip(GPI_Vector*,GPI_Function*,GPI_Vector*,GPI_Vector*[,GPI_Vector*])
GPI_zip	GPI_zip(GPI_Matrix*,GPI_Function*,GPI_Matrix*,GPI_Matrix*,GPI_index[,GPI_Vector*])
GPI_apply	GPI_apply(GPI_Vector*,GPI_Function*,GPI_Vector*,GPI_uint32[,GPI_Vector*])
GPI_apply	GPI_apply(GPI_Matrix*,GPI_Function*,GPI_Matrix*,GPI_uint32,GPI_index[,GPI_Vector*])
GPI_fixpt	GPI_fixpt(GPI_Vector*,GPI_Function*,GPI_Vector*[,GPI_Vector*])
GPI_fixpt	GPI_fixpt(GPI_Matrix*,GPI_Function*,GPI_Matrix*,GPI_index[,GPI_Vector*])
GPI_transpose	GPI_transpose(GPI_Matrix*,GPI_Matrix*)

Table 9: Derived methods prototypes for the C language bindings of GPI.

GPI method	C language prototype
GPI_innerp	GPI_innerp(GPI_Scalar*,GPI_Function*,GPI_Function*,GPI_Scalar,GPI_Vector*, GPI_Vector*[,GPI_Vector*])
GPI_innerp	GPI_innerp(GPI_Vector*,GPI_Function*,GPI_Function*,GPI_Vector*,GPI_Matrix*, GPI_Matrix*,GPI_index[,GPI_Vector*])
GPI_mxv	GPI_mxv(GPI_Vector*.GPI_Matrix*,GPI_Vector*,GPI_Function*,GPI_Function*[,GPI_Vector*]
GPI_vxm	GPI_vxm(GPI_Vector*.GPI_Vector*,GPI_Matrix*,GPI_Function*,GPI_Function*[,GPI_Vector*]
GPI_mxm	GPI_mxm(GPI_Matrix*.GPI_Matrix*,GPI_Matrix*,GPI_Function*,GPI_Function*)
GPI_successors	GPI_successors(GPI_Vector*,GPI_Matrix*,GPI_index)
GPI_predecessors	GPI_predecessors(GPI_Vector*,GPI_Matrix*,GPI_index)
GPI_reachability	GPI_reachability(GPI_Matrix*,GPI_Matrix*)

B Methods summary

All methods introduced in this report are listed alphabetically in Table 10.

Table 10: GPI Methods.

Function	Type	Page
GPI_apply	base	17
GPI_filter	base	14
GPI_filterneg	base	14
GPI_fixpt	base	17
GPI_indices	base	14
GPI_innerp	base	18
GPI_map	base	16
GPI_Matrix_col	accessor	12
GPI_Matrix_copy	entity	11
GPI_Matrix_delete	entity	12
GPI_Matrix_getElement	accessor	12
GPI_Matrix_ncols	accessor	12
GPI_Matrix_new	entity	11
GPI_Matrix_nrows	accessor	12
GPI_Matrix_row	accessor	12
GPI_Matrix_setElement	accessor	12
GPI_mxm	derived	19
GPI_mxv	derived	19
GPI_vxm	derived	19
GPI_predecessors	derived	20
GPI_reachability	derived	20
GPI_reduce	base	15
GPI_replicate	base	13
GPI_successors	derived	20
GPI_transpose	derived	18
GPI_Vector_copy	entity	11
GPI_Vector_delete	entity	11
GPI_Vector_getElement	accessor	13
GPI_Vector_new	entity	10
GPI_Vector_setElement	accessor	13
GPI_Vector_size	accessor	13
GPI_zip	base	16

C Example implementation of breadth first search

```
#include <stdio.h>
#include "gpi.h"

GPI_int32 BFS(GPI_Matrix *A, GPI_index s, GPI_Function1* visit)
/*
 * Given an adjacency matrix A and a source node s, performs a BFS traversal
 * and calls "visit" on the vertices visited (excluding source).
 */
{
  GPI_uint32 n; GPI_Matrix_nrows(&n, A);                  // n = number of vertices in graph
  GPI_Vector q, p, r, v;

  GPI_Vector_new(&q, GPI_int32, n);                       // int32[n] q
  GPI_replicate(&q, GPI_zero);                            // q = 0
  GPI_Vector_setElement(&q,s,GPI_one);                    // q[s] = 1 implies that node s is visited at current level
  GPI_Vector_new(&p, GPI_int32, n);                       // int32[n] p
  GPI_Vector_copy(&p,&q);                                 // p[s] = 1 implies that node s is already visited
  GPI_Matrix B; GPI_transpose(&B,A);                      // prepare transpose of adjacency matrix
  GPI_Vector_new(&r, GPI_int32, n);                       // int32[n] r
  GPI_Vector_new(&v, GPI_int32,n);                        // v = 0 .. n-1
  GPI_indices(&v);

/*
 * BFS traversal and visits the vertices.
 */
  bool done = false;                                      // done == true when BFS phase is complete
  do {
    GPI_replicate(&r, GPI_zero);
    GPI_mxv(&r,&B,&q,GPI_LOR,GPI_LAND);                   // r = (A^T)*q, finds all successors of nodes in q
    GPI_filter(&q,GPI_zero,&p,&r);                        // filter out nodes already visited
    GPI_zip(&p,GPI_LOR,&p,&q);                            // accumulate nodes visited
    GPI_map(&r,visit,&v,&q);                              // visit nodes in this level
    GPI_Scalar sum;
    GPI_reduce(&sum,GPI_LOR,GPI_zero,&q);                 // find if there is any node in q.
    done = (sum == GPI_zero);
  } while (!done);                                        // if there is no node in q, we are done.

  GPI_Vector_delete(&q); GPI_Vector_delete(&p);
  GPI_Vector_delete(&r); GPI_Vector_delete(&v);

  return 0;
}
```

D Example implementation of Brandes's betweenness centrality

```
#include <stdio.h>
#include "gpi.h"

GPI_int32 BC(GPI_Vector *delta, GPI_Matrix *A, GPI_index s)
/*
 * Given an adjacency matrix A and a source node s, compute BC-metric vector delta
 */
{
  GPI_uint32 n; GPI_Matrix_nrows(&n, A);                    // n = number of vertices in graph
  GPI_replicate(delta, GPI_zero);                           // delta = 0
  GPI_Matrix sigma; GPI_Matrix_new(&sigma, GPI_int32, n, n); // int32[n,n] sigma
  GPI_replicate(&sigma, GPI_zero);                          // sigma[d,k] = shortest path count to node k at level d
  GPI_Vector q; GPI_Vector_new(&q, GPI_int32, n);           // vector of path counts
  GPI_replicate(&q, GPI_zero);                              // q = 0
  GPI_Vector_setElement(&q, s, 1);                          // q[s] = 1
  GPI_Vector p; GPI_Vector_new(&p, GPI_int32, n);           // shortest path counts to all nodes so far
  GPI_Vector_copy(&p, &q);                                  // p = q

  /*
   * BFS phase
   */
  GPI_int32 d = 0;                                          // BFS level number
  bool done = false;                                        // done == true when BFS phase is complete
  do {
    GPI_Vector sigmad; GPI_Matrix_row(&sigmad,&sigma,d);    // row sigma[d,:] = q
    GPI_Vector_copy(&sigmad, &q);
    GPI_vxm(&q,&q,A,GPI_SUM,GPI_PROD);                      // q = path counts to nodes reachable from current level
    GPI_filter(&q,GPI_zero,&p,&q);                          // filter out nodes already visited
    GPI_zip(&p,GPI_SUM,&p,&q);                              // accumulate path counts on this level
    GPI_Scalar sum; GPI_reduce(&sum,GPI_SUM,GPI_zero,&q);   // sum path counts at this level
    done = (sum == GPI_zero);                               // if sum is zero, we are done
    d++;
  } while (!done);

  /*
   * BC computation phase
   */
  GPI_Vector t1; GPI_Vector_new(&t1,GPI_fp32,n);            // temporary vectors for computation
  GPI_Vector t2; GPI_Vector_new(&t2,GPI_fp32,n);
  GPI_Vector t3; GPI_Vector_new(&t3,GPI_fp32,n);
  GPI_Vector t4; GPI_Vector_new(&t4,GPI_fp32,n);
  for(int i=d-1; i>0; i--)
  {
    GPI_replicate(&t1, GPI_one);                            // add 1 to delta
    GPI_zip(&t1,GPI_SUM,&t1,delta);
    GPI_Vector sigmai; GPI_Matrix_row(&sigmai,&sigma,i);    // divide it by sigma for this level
    GPI_zip(&t2,GPI_DIV,&t1,&sigmai);
    GPI_mxv(&t3,A,&t2,GPI_SUM,GPI_PROD);                    // for each node, add contributions made by successors
    GPI_Vector sigmaim1; GPI_Matrix_row(&sigmaim1,&sigma,i-1); // multiply by corresponding sigma
    GPI_zip(&t4,GPI_PROD,&sigmaim1,&t3);
    GPI_zip(delta,GPI_SUM,delta,&t4);                       // accumulate into delta
  }

  GPI_Matrix_delete(&sigma);
  GPI_Vector_delete(&p); GPI_Vector_delete(&q);
  GPI_Vector_delete(&t1); GPI_Vector_delete(&t2); GPI_Vector_delete(&t3); GPI_Vector_delete(&t4);

  return 0;
}
```

E Example implementation of Prim's minimum spanning tree

```
#include <stdio.h>
#include <stdlib.h>
#include "gpi.h"

GPI_Scalar minnz(GPI_Scalar x, GPI_Scalar y)
{
   if (x == 0) return y;
   if (y == 0) return x;
   if (x > y) return y;
}

GPI_int32 MST(GPI_Vector *p, GPI_Matrix *A)
/* * * Given a weighted adjacency matrix A, compute the minimum spanning tree p
 * */
{
   GPI_size n; GPI_Matrix_nrows(&n, A);
   GPI_Vector s; GPI_Vector_new(&s, GPI_int32, n);
   GPI_replicate(&s,0);
   GPI_Vector_setElement(&s, 0, 1);
   GPI_replicate(p,0);

   GPI_Vector vi; GPI_Vector_new(&vi, GPI_int32, n);
   GPI_Vector vx; GPI_Vector_new(&vx, GPI_int32, n);
   GPI_Vector vv; GPI_Vector_new(&vv, GPI_int32, n);
   for(int i=1; i<n; i++)
   {
      GPI_Matrix M; GPI_Matrix_new(&M, GPI_int32, n, n);
      for(int j=0;j<n;j++)
      {
         GPI_Vector Mj; GPI_Matrix_col(&Mj, &M, j);
         GPI_Vector Aj; GPI_Matrix_col(&Aj, A, j);
         GPI_zip(&Mj,GPI_PROD,&s,&Aj);
      }
      for(int k=0;k<n;k++)
      {
         GPI_Vector Mk; GPI_Matrix_col(&Mk, &M, k);
         GPI_Scalar vik; GPI_Scalar vxk;
         GPI_reduce(&vik,minnz   ,0,&Mk);
         GPI_reduce(&vxk,GPLLOC,vik,&Mk);
         if (vik == 0) vxk = 0;
         GPI_Vector_setElement(&vi,k,vik);
         GPI_Vector_setElement(&vx,k,vxk);
      }
      GPI_filter(&vv,0,&s,&vi);
      GPI_Scalar min, j;
      GPI_reduce(&min,minnz,0,&vv);
      GPI_reduce(&j,GPLLOC,min,&vv);

      GPI_Scalar i; GPI_Vector_getElement(&i,&vx,j);
      GPI_Vector_setElement(&s,j,1);
      GPI_Vector_setElement(p,j,i);
   }

   GPI_Vector_delete(&s);
   GPI_Vector_delete(&vv); GPI_Vector_delete(&vi); GPI_Vector_delete(&vx);

   return 0;
}
```

F Example implementation of strongly connected components

```
#include <stdio.h>
#include "gpi.h"

GPI_int32 SCC(GPI_Vector* q, GPI_Matrix* A, GPI_Vector *b, GPI_int32 prefix)
/*
 * Given an adjacency matrix A, a vector b representing a subgraph (b[i] == true means vertex i is present
 * in the subset), and an integer prefix, returns a vector q of labels for the subgraph
 * so that all nodes belonging to a strongly connected component have a unique label
 */
{
  GPI_Vector_replicate(q,0);

  GPI_size n; GPI_Matrix_nrows(&n, A);                    // n = number of vertices in graph
  int k = -1; GPI_Scalar found;
  do { k++; GPI_Vector_getElement(&found,b,k); } while ((!found) && (k < n-1));  // find k such that b[k] == true
  if (!found) return 0;

  GPI_Vector p0; GPI_Vector_new(&p0,GPI_bool,n);          // p0 = predecessors of k
  GPI_predecessors(&p0,A,k);
  GPI_Vector s0; GPI_Vector_new(&s0,GPI_bool,n);          // s0 = successors of k
  GPI_successors(&s0,A,k);

  GPI_Vector p; GPI_Vector_new(&p,GPI_bool,n);            // p = p0 & b
  GPI_Vector s; GPI_Vector_new(&s,GPI_bool,n);            // s = s0 & b
  GPI_filterneg(&p,0,b,&p0);
  GPI_filterneg(&s,0,b,&s0);

  GPI_Vector bset; GPI_Vector_new(&bset,GPI_bool,n);      // bset = vertices that are both
  GPI_zip(&bset,GPILAND,&p,&s);                           // predessors and successors of k

  GPI_Vector pset; GPI_Vector_new(&pset,GPI_bool,n);      // pset = vertices that are only
  GPI_filter(&pset,0,&bset,&p);                           // predecessors of k

  GPI_Vector sset; GPI_Vector_new(&sset,GPI_bool,n);      // sset = vertices that are only
  GPI_filter(&sset,0,&bset,&s);                           // successors of k

  GPI_Vector blabels; GPI_Vector_new(&blabels,GPI_int32,n);
  GPI_Vector_replicate(&blabels,1+prefix*10);             // label the nodes in common
  GPI_filterneg(&blabels,0,&bset,&blabels);

  GPI_Vector plabels; GPI_Vector_new(&plabels,GPI_int32,n);
  SCC(&plabels,A,&pset,2+prefix*10);                      // components in predecessors subgraph

  GPI_Vector slabels; GPI_Vector_new(&slabels,GPI_int32,n);
  SCC(&slabels,A,&sset,3+prefix*10);                      // components in successors subgraph

  GPI_zip(q,GPI_SUM,&blabels,&plabels);
  GPI_zip(q,GPI_SUM,q,&slabels);                          // SCC labels of all nodes

  GPI_Vector_delete(&p); GPI_Vector_delete(&s);

  return 0;
}
```

G Graph generation

When studying and developing graph algorithms, it is desirable to test them with graphs of controlled size and characteristics. This is usually done through synthetic graphs produced through well defined processes. This section covers two commonly used graph generation algorithms: *the preferential attachment algorithm*, known to generate true power law distributions for vertices in-degree, and *the RMat algorithm*, known to offer control over the existence of communities in a graph.

G.1 The preferential attachment algorithm

Preferential attachment is a generative approach in which the end point of a new edge, or of an existing edge being moved, is selected from the existing vertices with a probability proportional to the existing vertices in-degree. Intuitively, it makes the highly connected vertices even more highly connected, giving rise to the power law distribution for the vertices in-degree, resulting in scale-free graphs.

As described in [2], the algorithm takes four parameters: m_0, m, p, and q. Parameter m_0 is the starting number of disconnected vertices. The algorithm then iterates on the following procedure until the desired number of vertices or edges has been reached. In each iteration, one of the following three actions is performed, with probabilities p, q, and $1 - p - q$ respectively.

1. A source vertex is chosen from the existing vertices randomly (uniform distribution over vertices), and an edge is added to a target vertex selected preferentially (the probably distribution function is in-degree of the vertex divided by the total number of edges in the graph). This is repeated m times for each iteration.
2. A source vertex and one of its edges are chosen randomly, and the target vertex of that edge is changed to a new target vertex chosen preferentially. This is repeated m times for each iteration.
3. A new vertex is created and m edges from it are added to one of the existing vertices selected preferentially.

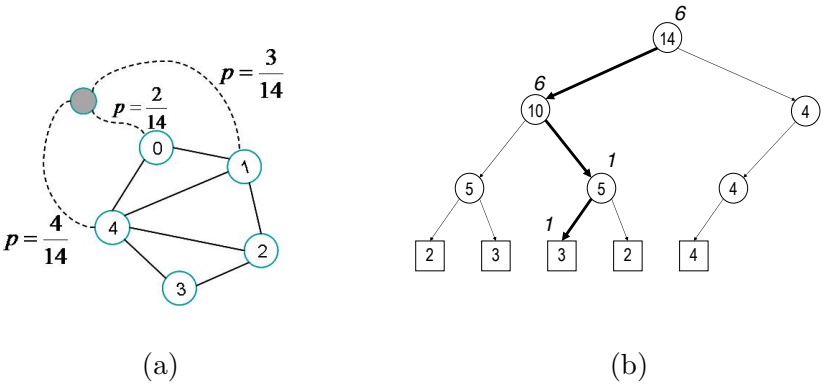
G.2 Implementation of preferential attachment algorithm

The key challenge in implementing the preferential attachment algorithm is the selection of a target vertex with probability proportional to its in-degree. Conceptually it is accomplished by maintaining an array c_i, $0 \leq i \leq n$, where n is the number of vertices, defined as:

$$c_0 = 0, \tag{1}$$

$$c_{i>0} = \sum_{j=0}^{i-1} d_j. \tag{2}$$

Where d_j is the degree for vertex j for $0 \leq j < n$, A random number x is taken using the uniform distribution over the range $[0, n)$. Vertex i is selected as the target of a new edge if $c_i \leq x < c_{i+1}$. A naive implementation of this approach would have quadratic complexity. So instead of maintaining the c_i explicitly we maintain a binary tree of partial sums. The leaf nodes of the tree are the in-degree values and each internal node has the sum of the in-degrees of all its leaf nodes, or equivalently, the sum of the values of its children. The binary tree is represented as a complete binary tree and therefore traversal towards the root or towards the leaf can be done with simple shift operation and increments.



(a) (b)

Figure 4: Probability for edge attachment (a), and partial sum tree (b).

Figure 4(a) illustrates a five node graph and a new shaded node from which we need to insert an edge to an existing node. The binary tree of partial sums is illustrated in Figure 4(b). Leaf nodes of the tree, in rectangles, correspond to the nodes of the graph, with their degree written inside the rectangle. Internal nodes have the sum of the degrees of all their leaf nodes written inside the circle. The dashed lines in Figure 4(a) illustrate the probability of attaching to various nodes.

To find the vertex i such that $c_i < x < c_{i+1}$, starting with an x taken from the uniform distribution over the range $[0, n)$ we start at the root and recursively take the left or right edge until we reach a leaf. The left edge is taken if x is less than the value of the node, otherwise the right edge is taken and value of x is decremented by the value of the left child. Figure 4(b) illustrates the selection of v_i when $x = 6$. The value of x is in italics outside the nodes and the path taken from root to leaf is shown in bold edges.

When a new edge is added to an end-vertex, we increment the value in the leaf node of the end-vertex of the edge, and all internal nodes on the path from the leaf to the root, including the root, in the tree of partial sums. Similarly, when an edge is removed from an end-vertex, we decrement the value in the leaf node of the end-vertex of the edge, and all internal nodes on the path from the leaf to the root, including the root, in the tree of partial sums.

G.3 The RMat algorithm

The RMat algorithm [3] offers better control over the existence of communities in a graph. The number of nodes is assumed to be an integral power of 2. It is a three parameter algorithm: A_0, B_0, and C_0. A derived parameter $D_0 = 1 - A_0 - B_0 - C_0$ is also used. For most use cases $B_0 = C_0$. The algorithm starts with an empty adjacency matrix and adds edges (entries in the adjacency matrix) until the desired number of edges have been added. The row and column indices for the edge to be inserted into the adjacency matrix are computed by recursively choosing the north-west, north-east, south-west and south-east quadrants with probabilities A_i, B_i, C_i, and D_i, respectively, where i is the depth of recursion. The probabilities are computed as follows. Let X stand for one of A, B, C, or D. We first compute

$$\bar{X}_{i+1} = X_i \cdot (0.95 + 0.1 \cdot \text{rand}(0,1)), X \in \{A, B, C, D\}, \tag{3}$$

where $\text{rand}(x, y)$ produces a uniformly distributed random number in the range $[x, y)$. We then normalize the intermediate results so that the probabilities add to 1:

$$X_{i+1} = \frac{\bar{X}_{i+1}}{A_{i+1} + B_{i+1} + C_{i+1} + D_{i+1}}, X \in \{A, B, C, D\}. \tag{4}$$

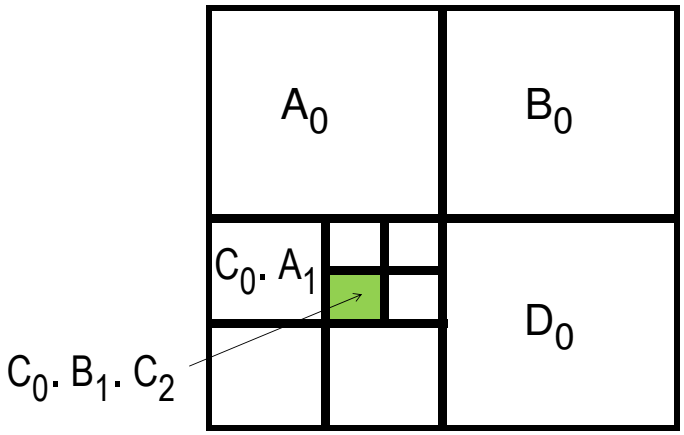


Figure 5: RMat edge insertion example.

Figure 5 illustrates the insertion of edge (5,2) as a result of quadrant selection outcomes associated with probabilities C_0, B_1, C_2.

The sequence of X_i generated in the above procedure can be viewed as columns of a $2 \times \log_2 N$ Boolean matrix. Then the rows correspond to the source and destination addresses of the edge to be inserted in a N node graph. Therefore the source and destination addresses can be computed by drawing X_i from uniform [0,3] distribution and left shifting its upper and lower bits into the source and destination addresses respectively, the two addresses having been initially set to 0.

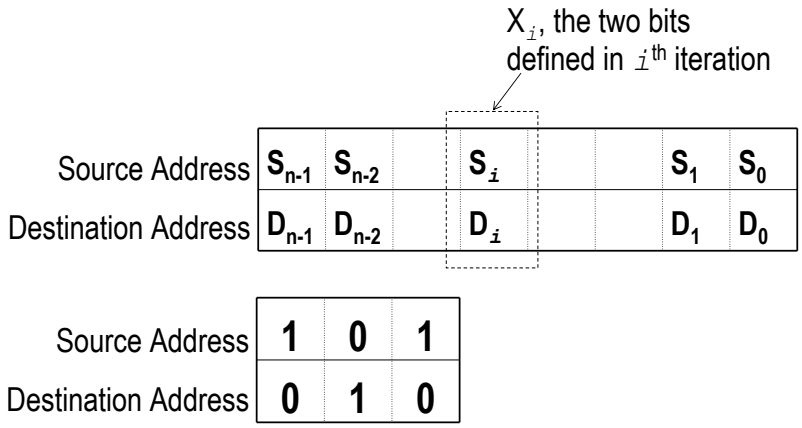


Figure 6: RMat selection of a node pair.

The selection of C_0, B_1, C_2 in Figure 5 corresponds to the random draws for X_i being 2, 3 and 2 as illustrated in Figure 6, and this draw of X_i corresponds to source and destination address being 5 and 2 respectively.