



# Scotch 7.0 Maintainer's Guide

(version 7.0.6)

François Pellegrini  
Université de Bordeaux & LaBRI, UMR CNRS 5800  
TadAAM team, INRIA Bordeaux Sud-Ouest  
351 cours de la Libération, 33405 TALENCE, FRANCE  
`francois.pellegrini@u-bordeaux.fr`

November 21, 2024

## **Abstract**

This document describes some internals of the LIBSCOTCH library.

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Coding style</b>	<b>3</b>
2.1	Typing . . . . .	3
2.1.1	Spacing . . . . .	3
2.1.2	Aligning . . . . .	3
2.1.3	Idiomatic specificities . . . . .	3
2.2	Indenting . . . . .	4
2.3	Comments . . . . .	4
<b>3</b>	<b>Naming conventions</b>	<b>5</b>
3.1	File inclusion markers . . . . .	5
3.2	Variables and fields . . . . .	5
3.3	Functions . . . . .	7
3.4	Array index basing . . . . .	8
<b>4</b>	<b>Structure of the libScotch library</b>	<b>8</b>
<b>5</b>	<b>Files and data structures</b>	<b>9</b>
5.1	Decomposition-defined architecture files . . . . .	9
<b>6</b>	<b>Adding a method to the libScotch library</b>	<b>10</b>
6.1	What to add . . . . .	10
6.2	Where to add . . . . .	11
6.3	Declaring the new method to the parser . . . . .	12
6.4	Adding the new method to the makefile . . . . .	13
<b>7</b>	<b>Data structure explanations</b>	<b>13</b>
7.1	Graph . . . . .	13
7.2	Hgraph . . . . .	16
7.3	Kgraph . . . . .	17
7.3.1	Mappings . . . . .	17
7.4	Mapping . . . . .	18
<b>8</b>	<b>Code explanations</b>	<b>20</b>
8.1	dgraphCoarsenBuild() . . . . .	20
8.1.1	Creating the fine-to-coarse vertex array . . . . .	20
8.2	dgraphFold() and dgraphFoldDup() . . . . .	21
8.2.1	dgraphFoldComm() . . . . .	22

## 1 Introduction

This document is a starting point for the persons interested in using SCOTCH as a testbed for their new partitioning methods, and/or willing to contribute to it by making these methods available to the rest of the scientific community.

Much information is missing. If you need specific information, please send an e-mail, so that relevant additional information can be added to this document.

## 2 Coding style

The SCOTCH coding style is now well established. Hence, potential contributors are requested to abide by it, to provide a global ease of reading while browsing the code, and to ease the work of their followers.

In this section, the numbering of the characters of each line is assumed to start from zero.

### 2.1 Typing

#### 2.1.1 Spacing

Expressions are like sentences, where words are separated by spaces. Hence, an expression like `if (n == NULL) {` reads: “if *n* is-equal-to NULL then”, with words separated by single spaces.

As in standard typesetting, there is no space after an opening parenthesis, nor before a closing one, because they are not words.

When it follows a keyword, an opening brace is always on the same line as the keyword (save for special cases, e.g. preprocessing macros between the keyword and the opening brace). This is meant to maximize the number of “useful readable lines” on the screen. However, closing braces are on a separate line, aligned with the indent of the line that contains the matching opening brace. This is meant to find in a glance the line that contains this opening brace.

Brackets are not considered as words: they are stuck both to the word on their left and the word on their right.

Reference and dereference operators “&” and “\*” are stuck to the word on their right. However, the multiplication operator “\*” counts as a word in arithmetic expressions.

Semicolons are always stuck to the word on their left, except when they follow an empty instruction, e.g., an empty loop body or an empty `for` field. Empty instructions are materialized by a single space character, which makes the semicolon separated from the preceding word. For instance: `for ( ; ; ) ;`.

Ternary operator elements “?” and “:” are considered as words and are surrounded by spaces. When the ternary construct spans across multiple lines, they are placed at the beginning of each line, before the expression they condition, and not at the end of the previous line.

#### 2.1.2 Aligning

When several consecutive lines contain similar expressions that are strongly connected, e.g. arguments of a `memAllocGroup()` routine, or assignments of multiple fields of the same structure(s), extra spaces can be added to align parts of the expressions. This is a matter of style and opportunity.

For instance, when consecutive lines contain function calls where opening parentheses are close to each other and their arguments overlap, open parentheses have to be aligned. However, when arguments do not overlap, alignment is not required (e.g., for `return` statements with small parameters).

#### 2.1.3 Idiomatic specificities

While, in C, `return` is a keyword which does not need parentheses around its argument, the SCOTCH coding style treats it as if it were a function call, thus requiring parentheses around its argument when it has one.

## 2.2 Indenting

Indenting is subject to the following rules:

- All indents are of two characters. Hence, starting from column zero, all lines start at even column numbers.
- Tabs are never used in the source code. If your text editor replaces chunks of spaces by tabs, it is your duty to disable this feature or to make sure to replace all tabs by spaces before the files are committed. Unwanted tabs are shown in red when performing a “`git diff`” prior to committing.

Condition bodies are always indented on the line below the condition statement. “`if`” statements are always placed at the beginning of a new line, except when used as an “`else if`” construct, in which the two keywords appear on the same line, separated by a single space.

Loop bodies are always indented on the line below the loop statement, except when the loop body is an empty instruction. In this case, the terminating semicolon is placed on the same line as the loop statement, after a single space.

## 2.3 Comments

All comments are C-style, that is, of the form “`/*...*/`”. C++-style comments should never be used.

There are three categories of comments: file comments, function/data structure comments, and line comments. Commenting is subject to the following rules:

- File comments are standard header blocks that must be copied as is. Hence, there is little to say about them. On top of each file should be placed a license header, which depends on the origin of the file.
- Block comments start with “`/*`” and end with “`*/`” on a separate subsequent line. Intermediate lines start with “`**`”. All these comment markers are placed at column zero. Comment text is separated from the comment markers by a single space character. Text in block comments is made of titles or of full sentences, that are terminated with a punctuation sign (most often a final dot).
- Line comments are of two types: structure definition line comments in header files, and code line comments.

Structure definition line comments in header files start with “`/*+`” and end with “`+*/`”. This is an old Doxygen syntax, which has been preserved over time. Code line comments start classically with “`/*`” and end with “`*/`”.

All these comments start at least at character 50. If the C code line is longer, comment lines start one character after the end of the line, after a single space. End comment markers are placed at least one character after the end of the comment text. When several line comments are present on consecutive lines, comment terminators are aligned to the farthest comment terminator.

Comment text always starts with an uppercase letter, and have no terminating punctuation sign. They are written in the imperative mode, and a positive form (no question asked).

Line comments for C pre-processing conditional macros (e.g. “`#else`” or “`#endif`”) are not subject to indentation rules. They start one character

after the keyword, and are not subject to end marker alignment, except when consecutive lines bear the same keyword (*i.e.*, a “`#endif`” statement).

### 3 Naming conventions

Data types, variables, structure fields and function names follow strict naming conventions. These conventions strongly facilitate the understanding of the meaning of the expressions, and prevent from coding mistakes. For instance, “`verttax[edgenum]`” would clearly be an invalid expression, as a vertex array cannot be indexed by an edge number. Hence, potential contributors are required to follow them strictly.

#### 3.1 File inclusion markers

File inclusion markers are `#define`’s which indicate that a given source file (either a “.c” source code file or a “.h” header file) has been already encountered.

To minimize risks of collisions with symbols of external libraries, file inclusion markers start with a prefix that represents the name of the project, followed by the name of the file in question (without its type suffix). While filenames can be long, this is not an issue since the length of the significant part of C preprocessor symbols is at least 63 characters<sup>1</sup>, thus longer than that of C identifiers, which is 32 characters. Header file marker identifiers are suffixed with “\_H”, while C source file markers have no suffix.

In order to further minimize risks of collisions, file inclusion markers should be placed in a file only when needed, that is, when effectively used as the parameter of a conditional inclusion statement within another source file.

The current project prefixes are:

- SCOTCH\_: the SCOTCH project itself;
- ESMUMPS\_: the ESMUMPS library, which is treated as a separate project to avoid conflicts with data structures and files that exist in both libraries, such as Graph’s.

#### 3.2 Variables and fields

Variables and fields of the sequential SCOTCH software are commonly built from a radical and a suffix. When contextualization is required, *e.g.*, the same kind of variable appear in two different objects, a prefix is added. In PT-SCOTCH, a second radical is commonly used, to inform on variable locality or duplication across processes.

Common radicals are:

- vert: vertex.
- velo: vertex load.
- vnoh: non-halo vertex, as used in the Hgraph structure.
- vnum: vertex number, used as an index to access another vertex structure. This radical typically relates to an array that contains the vertex indices, in some original graph, corresponding to the vertices of a derived graph (*e.g.*, an induced graph).

---

<sup>1</sup>See *e.g.* <https://gcc.gnu.org/onlinedocs/cpp/Implementation-limits.html>

- `vlbl`: user-defined vertex label (at the user API level).
- `edge`: edge (*i.e.*, arcs, in fact).
- `edlo`: edge (arc) load.
- `enoh`: non-halo edge (*i.e.*, arcs, in fact).
- `arch`: target architecture.
- `graf`: graph.
- `mesh`: mesh.

Common suffices are:

- `bas`: start “based” value for a number range; see the “`nnd`” suffix below. For number basing and array indexing, see Section 3.4.
- `end`: vertex end index of an edge (e.g., `vertend`, wrt. `vertnum`). The end suffix is a sub-category of the `num` suffix.
- `nbr`: number of instances of objects of a given radical type (e.g., `vertnbr`, `edgenbr`). They are commonly used within “un-based” loop constructs, such as: “`for (vertnum = 0; vertnum < vertnbr; vertnum ++)` ...”.
- `nnd`: end based value for a number range, commonly used for loop boundaries. Usually, `*nnd = *nbr+baseval`. For instance, `vertnnd = vertnbr+baseval`. They are commonly used in based loop constructs, such as: “`for (vertnum = baseval; vertnum < vertnnd; vertnum ++)` ...”. For local vertex ranges, e.g., within a thread that manages only a partial vertex range, the loop construct would be: “`for (vertnum = vertbas; vertnum < vertnnd; vertnum ++)` ...”.
- `num`: based or un-based number (index) of some instance of an object of a given radical type. For instance, `vertnum` is the index of some (graph) vertex, that can be used to access adjacency (`verttab`) or vertex load (`velotab`) arrays.  $0 \leq \text{vertnum} < \text{vertnbr}$  if the vertex index is un-based, and  $\text{baseval} \leq \text{vertnum} < \text{vertnnd}$  if the index is based, that is, counted starting from `baseval`.
- `ptr`: pointer to an instance of an item of some radical type (e.g., `grafptr`).
- `sum`: sum of several values of the same radical type (e.g., `velosum`, `edlosum`).
- `tab`: reference to the first memory element of an array. Such a reference is returned by a memory allocation routine (e.g., `memAlloc`) or allocated from the stack.
- `tax` (for “*table access*”): reference to an array that will be accessed using based indices. See Section 3.4.
- `tnd`: pointer to the based after-end of an array of items of radix type (e.g. `velotnd`). Variables of this suffix are mostly used as bounds in loops.
- `val`: value of an item. For instance, `baseval` is the indexing base value, and `veloval` is the load of some vertex, that may have been read from a file.

Common prefixes are:

- `src`: source, wrt. active. For instance, a source graph is a plain `Graph` structure that contains only graph topology, compared to enriched graph data structures that are used for specific computations such as bipartitioning.
- `act`: active, wrt. source. An active graph is a data structure enriched with information required for specific computations, e.g. a `Bgraph`, a `Kgraph` or a `Vgraph` compared to a `Graph`.
- `ind`: induced, wrt. original.
- `src`: source, wrt. active or target.
- `org`: original, wrt. induced. An original graph is a graph from which a derived graph will be computed, e.g. an induced subgraph.
- `tgt`: target.
- `coar`: coarse, wrt. fine (e.g. `coarvertnum`, as a variable that holds the number of a coarse vertex, within some coarsening algorithm).
- `fine`: fine, wrt. coarse.
- `mult`: multinode, for coarsening.

### 3.3 Functions

Like variables, routines of the SCOTCH software package follow a strict naming scheme, in an object-oriented fashion. Routines are always prefixed by the name of the data structure on which they operate, then by the name of the method that is applied to the said data structure. Some method names are standard for each class.

Standard method names are:

- `Alloc`: dynamically allocate an object of the given class. Not always available, as many objects are allocated on the stack as local variables.
- `Init`: initialization of the object passed as parameter.
- `Free`: freeing of the external structures of the object, to save space. The object may still be used, but it is considered as “empty” (e.g., an empty graph). The object may be re-used after it is initialized again.
- `Exit`: freeing of the internal structures of the object. The object must not be passed to other routines after the `Exit` method has been called.
- `Copy`: make a fully operational, independent, copy of the object, like a “clone” function in object-oriented languages.
- `Load`: load object data from stream.
- `Save`: save object data to stream.
- `View`: display internal structures and statistics, for debugging purposes.
- `Check`: check internal consistency of the object data, for debugging purposes. A `Check` method must be created for any new class, and any function that creates or updates an instance of some class must call the appropriate `Check` method, when compiled in debug mode.

### 3.4 Array index basing

The LIBSCOTCH library can accept data structures that come both from FORTRAN, where array indices start at 1, and C, where they start at 0. The start index for arrays is called the “base value”, commonly stored in a variable (or field) called `baseval`.

In order to manage based indices elegantly, most references to arrays are based as well. The “table access” reference, suffixed as “`tax`” (see Section 3.2), is defined as the reference to the beginning of an array in memory, minus the base value (with respect to pointer arithmetic, that is, in terms of bytes, times the size of the array cell data type). Consequently, for any array whose beginning is pointed to by `*tab`, we have `*tax = *tab - baseval`. Consequently `*tax[baseval]` always represents the first cell in the array, whatever the base value is. Of course, memory allocation and freeing operations must always operate on `*tab` pointers only.

In terms of indices, if the size of the array is `xxxxnbr`, then `xxxxnnd = xxxnbr + baseval`, so that valid indices `xxxxnum` always belong to the range `[baseval; vertnnd]`. Consequently, loops often take the form:

```
for (xxxxnum = baseval; xxxxnum < xxxxnnd; xxxxnum ++) {
    xxxxtax[xxxxnum] = ...;
}
```

## 4 Structure of the libScotch library

As seen in Section 3.3, all of the routines that comprise the LIBSCOTCH project are named with a prefix that defines the type of data structure onto which they apply and a prefix that describes their purpose. This naming scheme allows one to categorize functions as methods of classes, in an object-oriented manner.

This organization is reflected in the naming and contents of the various source files.

The main modules of the LIBSCOTCH library are the following:

- `arch`: target architectures used by the static mapping methods.
- `bgraph`: graph edge bipartitioning methods, hence the initial.
- `graph`: basic (source) graph handling methods.
- `hgraph`: graph ordering methods. These are based on an extended “halo” graph structure, thus for the initial.
- `hmesh`: mesh ordering methods.
- `kgraph`: k-way graph partitioning methods.
- `library`: API routines for the LIBSCOTCH library.
- `mapping`: definition of the mapping structure.
- `mesh`: basic mesh handling methods.
- `order`: definition of the ordering structure.
- `parser`: strategy parsing routines, based on the FLEX and BISON parsers.
- `vgraph`: graph vertex bipartitioning methods, hence the initial.

- `vmesh`: mesh node bipartitioning methods.

Every source file name is made of the name of the module to which it belongs, followed by one or two words, separated by an underscore, that describe the type of action performed by the routines of the file. For instance, for module `bgraph`:

- `bgraph.h` is the header file that defines the `Bgraph` data structure,
- `bgraph_bipart_fm.[ch]` are the files that contain the Fiduccia-Mattheyses-like graph bipartitioning method,
- `bgraph_check.c` is the file that contains the consistency checking routine `bgraphCheck` for `Bgraph` structures,

and so on. Every source file has a comments header briefly describing the purpose of the code it contains.

## 5 Files and data structures

User-manageable file formats are described in the SCOTCH user’s guide. This section contains information that are relevant only to developers and maintainers.

For the sake of portability, readability, and reduction of storage space, all the data files shared by the different programs of the SCOTCH project are coded in plain ASCII text exclusively. Although one may speak of “lines” when describing file formats, text-formatting characters such as newlines or tabulations are not mandatory, and are not taken into account when files are read. They are only used to provide better readability and understanding. Whenever numbers are used to label objects, and unless explicitly stated, **numberings always start from zero**, not one.

### 5.1 Decomposition-defined architecture files

Decomposition-defined architecture files are the way to describe irregular target architectures that cannot be represented as algorithmically-coded architectures.

Two main file formats coexist: the “`deco 0`” and “`deco 2`” formats. “`deco`” stands for “decomposition-defined architecture”, followed by the format number. The “`deco 1`” format is a compiled form of the “`deco 0`” format. We will describe it here.

The “`deco 1`” file format results from an  $O(p^2)$  preprocessing of the “`deco 0`” target architecture format. While the “`deco 0`” format contains a distance matrix between all pairs of terminal domains, which is consequently in  $\Theta(p^2/2)$ , the “`deco 1`” format contains the distance matrix between any pair of domains, whether they are terminal or not. Since there are roughly  $2p$  non-terminal domains in a target architecture with  $p$  terminal domains, because all domains form a binary tree whose leaves are the terminal domains, the distance matrix of a “`deco 1`” format is in  $\Theta(2p^2)$ , that is, four times that of the corresponding “`deco 0`” file.

Also, while the “`deco 0`” format lists only the characteristics of terminal domains (in terms of weights and labels), the “`deco 1`” format provides these for all domains, so as to speed-up the retrieval of the size, weight and label of any domain, whether it is terminal or not.

The “`deco 1`” header is followed by two integer numbers, which are the number of processors and the largest terminal number used in the decomposition, respectively (just as for “`deco 0`” files). Two arrays follow.

The first array has as many lines as there are domains (and not only terminal domains as in the case of “deco 0” files). Each of these lines holds three numbers: the label of the terminal domain that is associated with this domain (which is the label of the terminal domain of smallest number contained in this domain), the size of the domain, and the weight of the domain. The first domain in the array is the initial domain holding all the processors, that is, domain 1. The other domains in the array are the resulting subdomains, in ascending domain number order, such that the two subdomains of a given domain of number  $i$  are numbered  $2i$  and  $2i + 1$ .

The second array is a lower triangular diagonal-less matrix that gives the distance between all pairs of domains.

For instance, Figure 1 and Figure 2 show the contents of the “deco 0” and “deco 1” architecture decomposition files for  $UB(2, 3)$ , the binary de Bruijn graph of dimension 3, as computed by the `amk_grf` program.

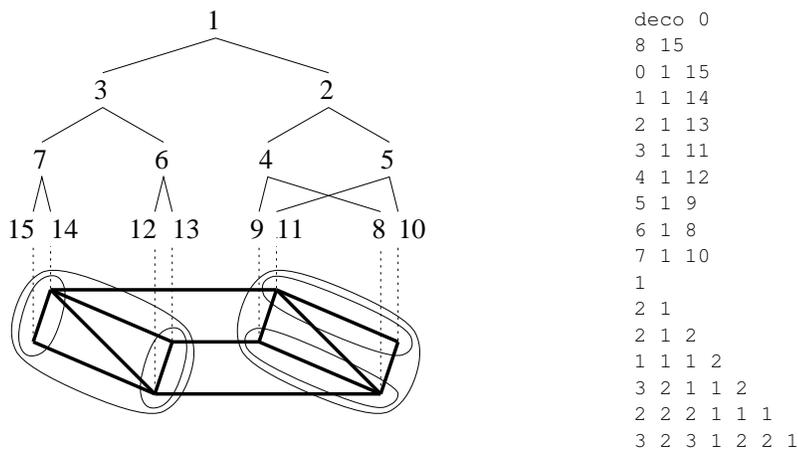


Figure 1: “deco 0” target decomposition file for  $UB(2, 3)$ . The terminal numbers associated with every processor define a unique recursive bipartitioning of the target graph.

## 6 Adding a method to the libScotch library

The `LIBSCOTCH` has been carefully designed so as to allow external contributors to add their new partitioning or ordering methods, and to use `SCOTCH` as a testbed for them.

### 6.1 What to add

There are currently six types of methods which can be added:

- k-way graph mapping methods, in module `kgraph`,
- graph bipartitioning methods by means of edge separators, in module `bgraph`, used by the mapping method by dual recursive bipartitioning, implemented in `kgraph_map_rb.[ch]`,
- graph ordering methods, in module `hgraph`,

```

deco          2 2 2 2 1 2 2 1
1            3 1 2 2 1 2 2 2
8 15        3 1 2 2 1 2 1 2
0 8 8       1 1 2 2 3 2 3 1
3 4 4       2 2 3 1 3 2 3 2
0 4 4       1 3 3 1 2 2 1 2
5 2 2       1 1 2 2 1 1 1 2
3 2 2       2 1 2 2 1 1 1 2
2 2 2       2 2 3 3 2 2 3 1
0 2 2       2 2 1 3 2 1 2 2
6 1 1       1 2 2 1 1 2 2 2
5 1 1       1 1 1 3 3 2 3 3
7 1 1       2 1 2 3 3 2 1 2
3 1 1       1
4 1 1
2 1 1
1 1 1
0 1 1

```

Figure 2: “deco 1” target decomposition file for UB(2,3), compiled with the `acpl` tool from the “deco 0” file displayed in Figure 1.

- graph separation methods by means of vertex separators, in module `vgraph`, used by the nested dissection ordering method implemented in `hgraph_order_nd.[ch]`,
- mesh ordering methods, in module `hmesh`,
- mesh separation methods with vertex separators, in module `vmesh`, used by the nested dissection ordering method implemented in `hmesh_order_nd.[ch]`.

Every method of these six types operates on instances of augmented graph structures that contain, in addition to the graph topology, data related to the current state of the partition or of the ordering. For instance, all of the graph bipartitioning methods operate on an instance of a `Bgraph`, defined in `bgraph.h`, and which contains fields such as `compload0`, the current load sum of the vertices assigned to the first part, `commload`, the load sum of the cut edges, etc.

In order to understand better the meaning of each of the fields used by some augmented graph or mesh structure, contributors can read the code of the consistency checking routines, located in files ending in `_check.c`, such as `bgraph_check.c` for a `Bgraph` structure. These routines are regularly called during the execution of the debug version of SCOTCH to ease bug tracking. They are time-consuming but proved very helpful in the development and testing of new methods.

## 6.2 Where to add

Let us assume that you want to code a new graph separation routine. Your routine will operate on a `Vgraph` structure, and thus will be stored in files called `vgraph_separate_xy.[ch]`, where `xy` is a two-letter reminder of the name of your algorithm. Look into the `LIBSCOTCH` source directory for already used codenames, and pick a free one. In case you have more than one single source file, use extended names, such as `vgraph_separate_xy-subname.[ch]`.

In order to ease your coding, copy the files of a simple and already existing method and use them as a pattern for the interface of your new method. Some methods have an optional parameter data structure, others do not. Browse through all existing methods to find the one that looks closest to what you want.

Some methods can be passed parameters at run time from the strategy string parser. These parameters can be of fixed types only. These types are:

- an integer (`int`) type,
- an floating-point (`double`) type,
- an enumerated (`char`) type : this type is used to make a choice among a list of single character values, such as “yn”. It is more readable than giving integer numerical values to method option flags,
- a strategy (SCOTCH `Strat` type) : a method can be passed a sub-strategy of a given type, which can be run on an augmented graph of the proper type. For instance, the nested dissection method in `hgraph_order.nd.c` uses a graph separation strategy to compute its vertex separators.

### 6.3 Declaring the new method to the parser

Once the new method has been coded, its interface must be known to the parser, so that it can be used in strategy strings. All of this is done in the module strategy method files, the name of which always end in `_st.[ch]`, that is, `vgraph_separate_st.[ch]` for the `vgraph` module. Both files are to be updated.

In the header file `*_st.h`, a new identifier must be created for the new method in the `StMethodType` enumeration type, preferably placed in alphabetical order.

In file `*_st.c`, there are several places to update. First, in the beginning of the module file, the header file of the new method, `vgraph_separate_xy.h` in this example, must be added in alphabetical order to the list of included method header files.

Then, if the new method has parameters, an instance of the method parameter structure must be created, which will hold the default values for the method. This is in fact a union structure, of the following form :

```
static union {
    VgraphSeparateXyParam    param;
    StratNodeMethodData      padding;
} vgraphseparatedefaultxy = { { ... } };
```

where the dots should be replaced by the list of default values of the fields of the `VgraphSeparateXyParam` structure. Note that the size of the `StratNodeMethodData` structure, which is used as a generic padding structure, must always be greater than or equal to the size of each of the parameter structures. If your new parameter structure is larger, you will have to update the size of the `StratNodeMethodData` type in file `parser.h`. The size of the `StratNodeMethodData` type does not depend directly on the size of the parameter structures (as could have been done by making it an union of all of them) so as to reduce the dependencies between the files of the library. In most cases, the default size is sufficient, and a test is added in the beginning of all method routines to ensure it is the case in practice.

Finally, the first two method tables must be filled accordingly. In the first one, of type `StratMethodTab`, one must add a new line linking the method identifier

to the character code used to name the method in strategy strings (which must be chosen among all of the yet unused letters), the pointer to the routine, and the pointer to the above default parameter structure if it exists (else, a `NULL` pointer must be given). In the second one, of type `StratParamTab`, one must add one line per method parameter, giving the identifier of the method, the type of the parameter, the name of the parameter in the strategy string, the base address of the default parameter structure, the actual address of the field in the parameter structure (both fields are required because the relative offset of the field with respect to the starting address of the structure cannot be computed at compile-time), and an optional pointer that references either the strategy table to be used to parse the strategy parameter (for strategy parameters) or a string holding all of the values of the character flags (for an enumerated type), this pointer being set to `NULL` for all of the other parameter types (integer and floating point).

## 6.4 Adding the new method to the makefile

Of course, in order to be compiled, the new method must be added to the `makefile` of the `libscotch` source directory. There are several places to update.

First, you have to create the entry for the new method source files themselves. The best way to proceed is to search for the one of an already existing method, such as `vgraph_separate_fm`, and copy it to the right neighboring place, preferably following the alphabetical order.

Then, you have to add the new header file to the dependency list of the module strategy method, that is, `vgraph_separate_st` for graph separation methods. Here again, search for the occurrences of string `vgraph_separate_fm` to see where it is done.

Finally, add the new object file to the component list of the `libscotch` library file.

Once all of this is done, you can recompile `SCOTCH` and be able to use your new method in strategy strings.

## 7 Data structure explanations

This section explains some of the data structures implemented in `SCOTCH` and `PT-SCOTCH`.

### 7.1 Graph

Graphs are the fundamental underlying data structures of all the algorithms implemented in `SCOTCH`. The `Graph` structure is the foundational data structure, from which subclasses will be derived, according to the specific needs of the `SCOTCH` modules. It is sometimes referred to as the *source graph* structure, with respect to the *target architecture* `Arch` onto which source graphs are to be mapped.

The `Graph` structure, being a foundational data structure, does not possess any variable fields related to actual computations, e.g., partition state variables or an execution context. Such fields will be found in *active* graphs, e.g., `Bgraph`, `Kgraph`, `Vgraph`.

A `Graph` is described by means of adjacency lists. These data are stored in arrays and scalars of type `SCOTCH_Num`, as shown in Figures 3 and 4. The `Graph` fields have the following meaning:

**baseval**  
 Base value for all array indexing.

**vertnbr**  
 Number of vertices in graph.

**edgenbr**  
 Number of arcs in graph. Since edges are represented by both of their ends, the number of edge data in the graph is twice the number of graph edges.

**verttax**  
 Based array of start indices in **edgetax** of vertex adjacency sub-arrays.

**vendtax**  
 Based array of after-last indices in **edgetax** of vertex adjacency sub-arrays. For any vertex  $i$ , with  $\text{baseval} \leq i < (\text{vertnbr} + \text{baseval})$ ,  $(\text{vendtax}[i] - \text{verttax}[i])$  is the degree of vertex  $i$ , and the indices of the neighbors of  $i$  are stored in **edgetax** from **edgetax**[**verttax**[ $i$ ]] to **edgetax**[**vendtax**[ $i$ ] - 1], inclusive.

When all vertex adjacency lists are stored in order in **edgetax**, it is possible to save memory by not allocating the physical memory for **vendtax**. In this case, illustrated in Figure 3, **verttax** is of size **vertnbr** + 1 and **vendtax** points to **verttax** + 1. This case is referred to as the “compact edge array” case, such that **verttax** is sorted in ascending order, **verttax**[**baseval**] = **baseval** and **verttax**[**baseval** + **vertnbr**] = (**baseval** + **edgenbr**).

**velotax**  
 Optional based array, of size **vertnbr**, holding the integer load associated with every vertex.

**edgetax**  
 Based array, of a size equal at least to  $(\max_i(\text{vendtax}[i]) - \text{baseval})$ , holding the adjacency array of every vertex.

**edlotax**  
 Optional based array, of a size equal at least to  $(\max_i(\text{vendtax}[i]) - \text{baseval})$ , holding the integer load associated with every arc. Matching arcs should always have identical loads.

Dynamic graphs can be handled elegantly by using the **vendtax** array. In order to dynamically manage graphs, one just has to allocate **verttax**, **vendtax** and **edgetax** arrays that are large enough to contain all of the expected new vertex and edge data. Original vertices are labeled starting from **baseval**, leaving free space at the end of the arrays. To remove some vertex  $i$ , one just has to replace **verttax**[ $i$ ] and **vendtax**[ $i$ ] with the values of **verttax**[**vertnbr** - 1] and **vendtax**[**vertnbr** - 1], respectively, and browse the adjacencies of all neighbors of former vertex **vertnbr** - 1 such that all (**vertnbr** - 1) indices are turned into  $i$ s. Then, **vertnbr** must be decremented.

To add a new vertex, one has to fill **verttax**[**vertnbr** - 1] and **vendtax**[**vertnbr** - 1] with the starting and end indices of the adjacency sub-array of the new vertex. Then, the adjacencies of its neighbor vertices must also be updated to account for it. If free space had been reserved at the end of each of the neighbors, one just has to increment the **vendtax**[ $i$ ] values of every neighbor  $i$ , and add the index of the new vertex at the end of the adjacency sub-array. If the sub-array cannot be extended,

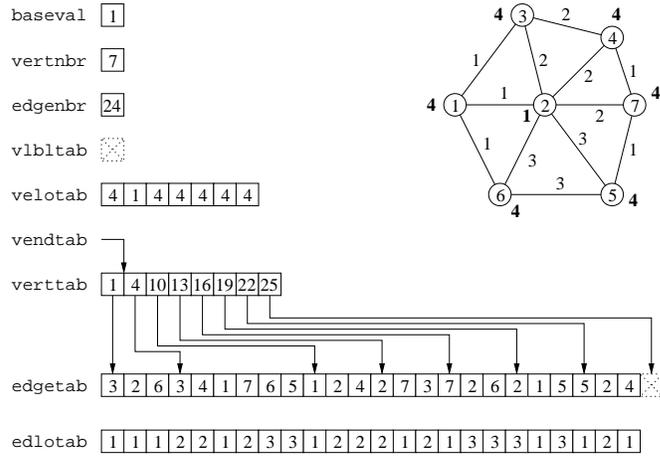


Figure 3: Sample graph and its description using a compact edge array. Numbers within vertices are vertex indices, bold numbers close to vertices are vertex loads, and numbers close to edges are edge loads. Since the edge array is compact, **verttax** is of size **vertnbr** + 1 and **vendtax** points to **verttax** + 1.

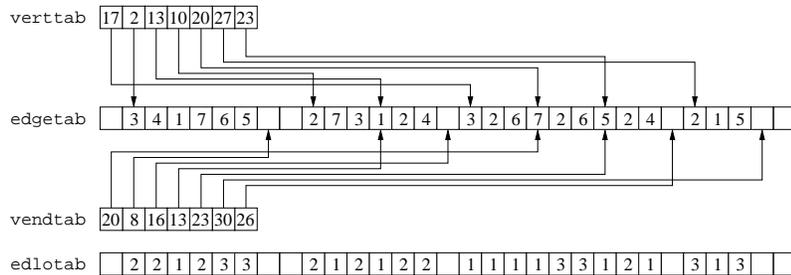


Figure 4: Adjacency structure of the sample graph of Figure 3 with disjoint edge and edge load arrays. Both **verttax** and **vendtax** are of size **vertnbr**. This allows for the handling of dynamic graphs, the structure of which can evolve with time.

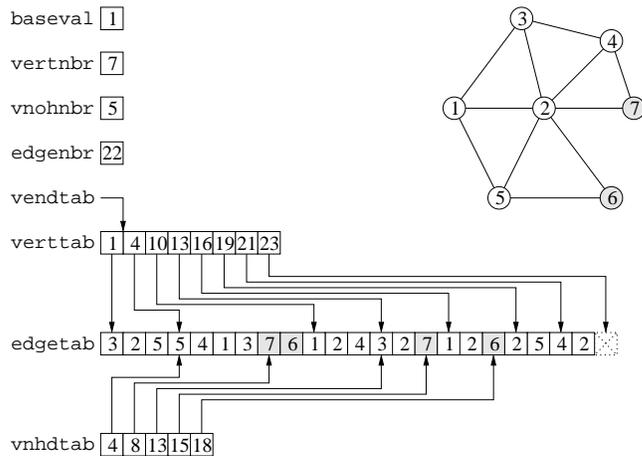


Figure 5: Sample halo graph and its description using a compact edge array. Numbers within vertices are vertex indices. Greyed values are indices of halo vertices. Halo vertices have the highest indices in the graph, and are placed last in the adjacency sub-arrays of each non-halo vertex.

then it has to be copied elsewhere in the edge array, and both `verttax[i]` and `vendtax[i]` must be updated accordingly. With simple housekeeping of free areas of the edge array, dynamic arrays can be updated with as little data movement as possible.

## 7.2 Hgraph

The `Hgraph` structure holds all the information necessary to represent and perform computations on a *halo* graph. This term refers to graphs some vertices of which are kept to preserve accurate topological information, but are usually not subject to actual computations. These *halo vertices* are collectively referred to as the *halo* of the graph. Halo graphs are notably used in sparse matrix reordering, where, in the process of nested dissection, a graph is cut into three pieces: a vertex separator, and two separated parts. Each of these parts must preserve the real degree information attached to all their vertices, including those next to the separator. If halo graphs were not used, the degrees of these vertices would appear smaller than what they really are in the whole graph. Preserving accurate degree information is essential for algorithms such as the *minimum degree* vertex ordering method. Some vertex separation algorithms also aim at balancing halo vertices; in this case, separators will be computed on halos, but this information will not be preserved once a separator has been computed on the regular vertices.

Halo graphs exhibit specific structural and topological properties, illustrated in Figure 5. In order to distinguish easily halo vertices from regular vertices and write efficient algorithms, halo vertices have the highest vertex indices in the graph. Because the degrees of halo vertices need not be preserved, no edges connect two halo vertices; the adjacency of halo vertices is only made of regular vertices. Also, in the adjacency arrays of regular vertices, all non-halo vertices are placed before halo vertices. All these properties allow one to easily induce the non-halo graph from some halo graph, without having to create new adjacency arrays. An additional vertex index array is present just for this purpose.

Halo graph fields have the following meaning:

`s` Underlying source graph that contains all regular and halo vertices. This is where to search for fields such as `baseval`, `vertnbr`, `vertnnd`, `verttax`, `vendtax`, etc.

`vnohnbr`

Number of non-halo vertices in graph. Hence,  $0 \leq \text{vnohnbr} \leq \text{s.vertnbr}$ .

`vnhdtax`

Array of after-last indices in `s.edgetax` of non-halo vertex adjacency sub-arrays. Since this information only concerns non-halo vertices, `vnhdtax` is of size `vnohnbr`, not `vertnbr`. For any non-halo vertex  $i$ , with `baseval`  $\leq i < (\text{vnohnbr} + \text{baseval})$ , the indices of the non-halo neighbors of  $i$  are stored in `s.edgetax` from `s.edgetax[s.verttax[i]]` to `s.edgetax[vnhdtax[i] - 1]`, inclusive, and its halo neighbors are stored from `s.edgetax[vnhdtax[i]]` to `s.edgetax[s.vendtax[i] - 1]`, inclusive.

`vnlosum`

Sum of non-halo vertex loads. Hence,  $0 \leq \text{vnlosum} \leq \text{s.velosum}$ .

`enohnbr`

Number of non-halo arcs in graph. Hence,  $0 \leq \text{enohnbr} \leq \text{s.edgenbr}$ .

## 7.3 Kgraph

The `Kgraph` structure holds all the information necessary to compute a  $k$ -way (re)mapping of some graph onto a target architecture. Consequently, it contains a `Graph`, defined as field `s`, and a reference to an `Arch`, through the field `m.archptr`, as well as two `Mapping` structures: one for the current mapping to compute, and one to store the old mapping from which to remap. Additional information comprise data to model the cost of remapping, and data associated with the state and cost of the current mapping: list of frontier vertices, load of each partition domain, plus the execution context for multi-threading execution.

The `Graph` structure is internal to the `Kgraph` because every new `Kgraph` contains a different graph topology (e.g., a band graph or a coarsened graph). The `Arch` is accessed by reference because it is constant data which can be shared by many `Kgraphs`. For the sake of consistency, the `grafptr` fields of each mapping `m` and `r.m` must point to `&s`, while their two `archptr` fields must point to the same target architecture. This redundancy is the price to pay for lighter memory management.

### 7.3.1 Mappings

The `domnorg` field, which must contain a valid domain in the architecture `m.archptr`, is the starting point for the  $k$ -way mapping. This domain may be smaller than the full architecture when parallel partitioning is performed: in this case, each process may receive a separate subgraph and sub-architecture to work on.

Each of the two mappings has its own specificities. The current mapping, defined as field `m`, is never incomplete: all the cells of its `m.parttax` array are non-negative values that index a valid domain in the domain array `m.domntab`. These domains are all subdomains of the architecture referenced through field `m.archptr`. More restrictively, the domains attached to non-fixed vertices must be included in `domnorg`, which may be smaller.

The current mapping evolves with time, according to the various algorithms that the user can activate in the strategy string. These algorithms will create derived `Kgraph`s (e.g., band graphs or coarsened graphs), to which mapping methods will be applied, before the result is ported back to their parent `Kgraph`. Depending on the kind of the derived graph, the `m.parttax` array may be specific, but the `m.domntab` array will always be ported back as is. Consequently, in order to save memory copying, the policy which is implemented is that the derived `Kgraph` gets the pointer to the `m.domntab` of its parent, while the latter is set to `NULL`. The derived graph can therefore reallocate the array whenever needed, without the risk of an old, invalid, pointer being kept elsewhere. Then, when the processing of the derived `Kgraph` ends, the most recent pointer is copied back to the `m.domntab` field of the parent graph, and the `m.parttax` array is updated accordingly, after which the derived `Kgraph` can be destroyed without freeing the pointer.

The old mapping, defined as field `r.m`, may contain incomplete mapping information: some of the cells of its `r.m.parttax` array may be equal to `-1`, to indicate that no prior mapping information is available (e.g., when the vertex did not exist in the previous mapping). Since old mappings do not change, the `r.m.domntab` field can be shared among all derived `Kgraph`s. It is protected from double memory freeing by not setting the `MAPPINGFREEDOMN` flag in field `r.m.flagval`.

## 7.4 Mapping

The `Mapping` structure defines how individual vertices of a `Graph` are mapped individually onto (parts of) an `Arch`. A mapping is said *complete* if all source graph vertices are assigned to terminal target domains, *i.e.*, individual vertices of the target architecture, or *partial* if at least one of the source graph vertices is assigned to a target domain that comprises more than one vertex. In the course of the graph mapping process, the destination of source vertices are progressively refined, from an initial target domain that usually describes the whole of the target architecture, to terminal domains.

Since `ArchDom`, the data structure that describes target architecture domains, is big and costly to handle (e.g., to compare if two `ArchDoms` are identical), the handling of domains in mapping is indirect: in the `part` array `parttax`, each vertex is assigned an integer domain index that refers to a domain located in the `domain` array `domntab`. Hence, when two graph vertices have the same index in `parttax`, they belong to the same domain and induce no communication cost. However, the opposite is false: two vertices may have a different index in `parttax` and yet belong to the same target domain. This is for instance the case when one of the vertices is a fixed vertex that has been set to a specific terminal domain at initialization time, and one of its neighbors is successively mapped to smaller and smaller subdomains that eventually amount to the same terminal domain.

In the case of a remapping, the mapping information regarding the former placement of the vertices may be incomplete, e.g., because the vertex did not exist before. Such a mapping is said to be *incomplete*. It is characterized by the fact that some cells of the `parttax` array are equal to `-1`, to indicate an unknown terminal domain number. To allow for this, the mapping must have the `MAPPING INCOMPLETE` flag set. Incomplete mappings are only valid when holding remapping information; new mappings being computed must have all their `parttax` cells set with non-negative values that point to valid domains in the `domntab` array. New mappings can therefore only be partial or complete.

When a mapping is initialized, all `parttax` values for non-fixed vertices

are set to index 0, and `domntab[0]` is set to the root domain for the mapping. In the general case for centralized mapping, the initial domain is equal to `archDomFrst(archptr)`. However, when a centralized mapping process is launched as a part of a distributed mapping process, the initial domain may be a subset of the whole target architecture.

There is no obligation for the `domntab` array to contain only one instance of some target domain. On the contrary, as described above, the same domain may appear at least twice: once for fixed vertices, and once for non-fixed vertices on which mapping algorithms are applied. However, for efficiency reasons (e.g., avoiding to compute vertex distances that are equal to zero), it is preferable that duplicate domains are avoided in the `domntab` array. This is the case by nature with recursive bipartitioning, as the domains associated with branches of the bipartitioning tree are all distinct.

Making the distinction between fixed and non-fixed vertices, which is relevant to mapping algorithms, is not in the scope of the `Mapping` data structure, which only represents a global state. This is why no data related to fixed vertices is explicitly present in the mapping itself (it may be found, e.g., in the `Kgraph` data structure). However, for handling fixed vertices in an efficient way, the semantics of the `Mapping` data structure is that all domains that are associated with fixed vertices must be placed first in the `domntab` array. The purpose of this separation is because, when the imbalance of a mapping is computed, the loads of non-fixed vertices that belong to some (partial) domain and of fixed vertices that belong to domains that are subdomains of this domain have to be aggregated. This aggregation procedure is made easier if both types of domains are kept separate. For efficiency reasons, fixed domains should appear only once in the fixed part of `domntab`.

The `Mapping` structure is mainly used within the `Kgraph` structure, which contains two instances of it: one for the current mapping to be computed, and one for the old mapping, in the case of remapping. The building of a `Kgraph` from another one (e.g., when creating a band graph or a coarsened graph) may lead to situations in which some `Mapping` arrays may be re-used, and thus should not be freed when the derived `Mapping` is freed. This is why the `Mapping` structure contains flags to record whether its arrays should be freed or not. These flags are the following:

- `MAPPINGFREEDOMN`: set if the domain array has to be freed when the mapping is freed. A common case for sharing the domain array is when a coarser `Kgraph` is computed: the domain array of the coarse old mapping can re-use that of the fine old mapping.
- `MAPPINGFREEPART`: set if the part array has to be freed when the mapping is freed. A common case for sharing the part array is when the user part array is kept as the part array for the initial `Kgraph` current mapping structure.

The main fields of the `Mapping` data structure are the following:

- `flagval`: set of flags that defines whether the `parttax` and `domntab` have to be freed on exit.
- `grafptr`: pointer to the `Graph` associated with the mapping, that gives access to the base value `grafptr->baseval` and the number of source vertices `grafptr->vertnbr`.

- `archptr`: pointer to the `Arch` associated with the mapping, that is necessary to perform all distance computations on the mapping.
- `parttax`: based array of `Anums`, of size `grafptr->vertnbr`, that provides the index of the target domains onto which all graph vertices are currently mapped. Indices are un-based.
- `domntab`: un-based array of `ArchDoms`, of size `domnmax`, that stores the target domains to which source graph vertices are indirectly associated through the `parttax` array.
- `domnnbr`: number of target domain slots currently used in `domntab`. After a mapping is initialized,  $1 \leq \text{domnnbr} < \text{domnmax}$ , because source graph vertices must be associated to some domain, hence `domntab` should at least contain one domain.
- `domnnbr`: number of target domain slots currently used in `domntab`.
- `domnmax`: size of the `domntab` array.
- `mutedat`: when multi-threading is activated, allows to create critical sections to update the mapping data in a thread-safe manner.

## 8 Code explanations

This section explains some of the most complex algorithms implemented in SCOTCH and PT-SCOTCH.

### 8.1 `dgraphCoarsenBuild()`

The `dgraphCoarsenBuild()` routine creates a coarse distributed graph from a fine distributed graph, using the result of a distributed matching. The result of the matching is available on all MPI processes as follows:

- `coardat.multlocnbr`: the number of local coarse vertices to be created;
- `coardat.multloctab`: the local multinode array. For each local coarse vertex to be created, it contains two values. The first one is always positive, and represents the global number of the first local fine vertex to be mated. The second number can be either positive or negative. If it is positive, it represents the global number of the second local fine vertex to be mated. If it is negative, its opposite, minus two, represents the local edge number pointing to the remote vertex to be mated; `coardat.procgsttax`: array (restricted to ghost vertices only) that records on which process is located each ghost fine vertex.

#### 8.1.1 Creating the fine-to-coarse vertex array

In order to build the coarse graph, one should create the array that provides the coarse global vertex number for all fine vertex ends (local and ghost). This information will be stored in the `coardat.coargsttax` array.

Hence, a loop on local multinode data fills `coardat.coargsttax`. The first local multinode vertex index is always local, by nature of the matching algorithm. If the second vertex is local too, `coardat.coargsttax` is filled instantly. Else,

a request for the global coarse vertex number of the remote vertex is forged, in the `vsnddattab` array, indexed by the current index `coarsndidx` extracted from the neighbor process send index table `nsndidx`. Each request comprises two numbers: the global fine number of the remote vertex for which the coarse number is sought, and the global number of the coarse multinode vertex into which it will be merged.

Then, an all-to-all-v data exchange by communication takes place, using either the `dgraphCoarsenBuildPtop()` or `dgraphCoarsenBuildColl()` routines. Apart from the type of communication they implement (either point-to-point or collective), these routines do the same task: they process the pairs of values sent from the `vsnddattab` array. For each pair (the order of processing is irrelevant), the `coargsttax` array of the receiving process is filled-in with the global multinode value of the remotely mated vertex. Hence, at the end of this phase, all processes have a fully valid local part of the `coargsttax` array; no value should remain negative (as set by default). Also, the `nrcvidxtab` array is filled, for each neighbor process, of the number of data it has sent. This number is preserved, as it will serve to determine the number of adjacency data to be sent back to each neighbor process.

Then, data arrays for sending edge adjacency are filled-in. The `ercvdsptab` and `ercvnttab` arrays, of size `procglbnbr`, are computed according to the data stored in `coardat.dcntglbtab`, regarding the number of vertex- and edge-related data to exchange.

By way of a call to `dgraphHaloSync()`, the ghost data of the `coargsttax` array are exchanged.

Then, `edgelocnbr`, an upper bound on the number of local edges, as well as `ercvdatsiz` and `esnddatsiz`, the edge receive and send array sizes, respectively.

Then, all data arrays for the coarse graph are allocated, plus the main adjacency send array `esnddsptab`, its receive counterpart `ercvdattab`, and the index send arrays `esnddsptab` and `esndcnttab`, among others.

Then, adjacency send arrays are filled-in. This is done by performing a loop on all processes, within which only neighbor processes are actually considered, while index data in `esnddsptab` and `esndcnttab` is set to 0 for non-neighbor processes. For each neighbor process, and for each vertex local which was remotely mated by this neighbor process, the vertex degree is written in the `esnddsptab` array, plus optionally its load, plus the edge data for each of its neighbor vertices: the coarse number of its end, obtained through the `coargsttax` array, plus optionally the edge load. At this stage, two edges linking to the same coarse multinode will not be merged together, because this would have required a hash table on the send side. The actual merging will be performed once, on the receive side, in the next stage of the algorithm.

## 8.2 `dgraphFold()` and `dgraphFoldDup()`

The `dgraphFold()` routine creates a “folded” distributed graph from the input distributed graph. The folded graph is such that it spans across only one half of the processing elements of the initial graph (either the first half, or the second half). The purpose of this folding operation is to preserve a minimum average number of vertices per processing element, so that communication cost is not dominated by message start-up time. In case of an odd number of input processing elements, the first half of them is always bigger than the second.

The `dgraphFoldDup()` routine creates two folded graphs: one for each half. Hence, each processing element hosting the initial graph will always participate in

hosting a new graph, which will depend on the rank of the processing element. When the MPI implementation supports multi-threading, and multi-threading is activated in SCOTCH, both folded graphs are created concurrently.

The folding routines are based on the computation of a set of (supposedly efficient) point-to-point communications between the *sender processes*, which will not retain any graph data, and the *receiver processes*, which will host the folded graph. However, in case of unbalanced vertex distributions, overloaded receiver processes (called *sender receiver processes*) may also have to send their extra vertices to underloaded receiver processes. A receiver process may receive several chunks of vertex data (including their adjacency) from several sender processes. Hence, folding amounts to a redistribution of vertex indices across all receiver processes. In particular, end vertex indices have to be renumbered according to the global order in which the chunks of data are exchanged. This is why the computation of these exchanges, by way of the `dgraphFoldComm()` routine, has to be fully deterministic and reproducible across all processing elements, to yield consistent communication data. The result of this computation is a list of point-to-point communications (either all sends or receives) to be performed by the calling process, and an array of sorted global vertex indices, associated with vertex index adjustment values, to convert global vertex indices in the adjacency of the initial graph into global vertex indices in the adjacency of the folded graph. This array can be used, by way of dichotomy search, to find the proper adjustment value for any end vertex number.

To date, the `dgraphRedist()` routine is not based on a set of point-to-point communications, but collectives. It could well be redesigned to re-use the mechanisms implemented here, with relevant code factorization.

### 8.2.1 `dgraphFoldComm()`

The `dgraphFoldComm()` routine is at the heart of the folding operation. It computes the sets of point-to-point communications required to move vertices from the sending half of processing elements to the receiving half, trying to balance the folded graph as much as possible in terms of number of vertices. For receiver processes, it also computes the data needed for the renumbering of the adjacency arrays of the graph chunks received from sender (or sender receiver) processes.

It is to be noted that the end user and the SCOTCH algorithms may have divergent objectives regarding balancing: in the case of a weighted graph representing a computation, where some vertices bear a higher load than others, the user may want to balance the load of its computations, even if it results in some processing elements having less vertices than others, provided the sums of the loads of these vertices are balanced across processing elements. On the opposite, the algorithms implemented in SCOTCH operate on the vertices themselves, irrespective of the load values that is attached to them (save for taking them into account for computing balanced partitions). Hence, what matters to SCOTCH is that the number of vertices is balanced across processing elements. Whenever SCOTCH is provided with an unbalanced graph, it will try to rebalance it in subsequent computations (e.g., folding). However, the bulk of the work, on the initial graph, will be unbalanced according to the user's distribution.

During a folding onto one half of the processing elements, the processing elements of the other half will be pure senders, that need to dispose of all of their vertices and adjacency. Processing elements of the first half will likely be receivers, that will take care of the vertices sent to them by processing elements of the other half. However, when a processing element in the first half is overloaded, it may behave

as a sender rather than a receiver, to dispose of its extra vertices and send it to an underloaded peer.

The essential data that is produced by the `dgraphFoldComm()` routine for the calling processing element is the following:

- `commmax`: the maximum number of point-to-point communications that can be performed by any processing element. The higher this value, the higher the probability to spread the load of a highly overloaded processing element to (underloaded) receivers. In the extreme case where all the vertices are located on a single processing element,  $(\text{procglbnbr} - 1)$  communications would be necessary. To prevent such a situation, the number of communications is bounded by a small number, and receiver processing elements can be overloaded by an incoming communication. The algorithm strives to provide a *feasible* communication scheme, where the current maximum number of communications per processing element suffices to send the load of all sender processing elements. When the number of receivers is smaller than the number of senders (in practice, only by one, in case of folding from an odd number of processing elements), at least two communications have to take place on some receiver, to absorb the vertices sent. The initial maximum number of communications is defined by `DGRAPHFOLDCOMMNBR`;
- `commtypval`: the type of communication and processing that the processing element will have to perform: either as a sender, a receiver, or a sender receiver. Sender receivers will keep some of their vertex data, but have to send the rest to other receivers. Sender receivers do send operations only, and never receive data from a sender;
- `commdattab`: a set of slots, of type `DgraphFoldCommData`, that describe the point-to-point communications that the processing element will initiate on its side. Each slot contains the number of vertices to send or receive, and the target or source process index, respectively;
- `commvrctab`: a set of values associated to each slot in `commdattab`, each of which contains the global index number of the first vertex of the graph chunk that will be transmitted;
- `proccnttab`: for receiver processes only, the count array of same name of the folded distributed graph structure;
- `vertadjnbr`: for receiver processes only, the number of elements in the dichotomy array `vertadjtab`;
- `vertadjtab`: a sorted array of global vertex indices. Each value represent the global start index of a graph chunk that will be exchanged (or which will remain in place on a receiver processing element);
- `vertdlftab`: the value which has to be added to the indices of the vertices in the corresponding chunk represented in `vertadjtab`. This array and the latter serve to find, by dichotomy, to which chunk an end vertex belongs, and modify its global vertex index in the edge array in the receiver processing element. Although `vertadjtab` and `vertdlftab` contain strongly related information, they are separate arrays, for the sake of memory locality. Indeed, `vertadjtab` will be subject to a dichotomy search, involving many memory reads, before the proper index is found and a single value is retrieved from the `vertdlftab` array.

The first stage of the algorithm consists in sorting a global process load array in ascending order, in two parts: the sending half, and the receiving half. These two sorted arrays will contain the source information which the redistribution algorithm will use. Because the receiver part of the sort array can be modified by the algorithm, it is recomputed whenever `commmax` is incremented. It is the same for `sortsndbas`, the index of the first non-empty sender in the sort array.

In a second stage, the algorithm will try to compute a valid communication scheme for vertex redistribution, using as many as `commmax` communications (either sends or receives) per processing element. During this outermost loop, if a valid communication scheme cannot be created, then `commmax` is incremented and the communication scheme creation algorithm is restarted. The initial value for `commmax` is `DGRAPHFOLDCOMMNR`.

The construction of a valid communication scheme is performed within an intermediate loop. At each step, a candidate sender process is searched for: either a sender process which has to dispose of all of its vertices, or an overloaded receiver process, depending on which has the biggest number of vertices to send. If candidate senders can no longer be found, the stage has succeeded with the current value of `commmax`; if a candidate sender has been found but a candidate receiver has not, the outermost loop is restarted with an incremented `commmax` value, so as to balance loads better.

Every time a sender has been found and one or more candidate receivers exist, an inner loop creates as many point-to-point communications as to spread the vertices in chunks, across one or more available receivers, depending on their capacity (*i.e.*, the number of vertices they can accept). If the selected sender is a sender receiver, the inner loop will try to interleave small communications from pure senders with communications of vertex chunks from the selected sender receiver. The purpose of this interleaving is to reduce the number of messages per process: a big message from a sender receiver is likely to span across several receivers, which will then perform only a single receive communication. By interleaving a small communication on each of the receivers involved, the latter will only have to perform one more communication (*i.e.*, two communications only), and the interleaved small senders will be removed off the list, reducing the probability that afterwards many small messages will sent to the same (possibly eventually underloaded) receiver.

In a third stage, all the data related to chunk exchange, which was recorded in a temporary form in the `vertadjtab`, `vertdlftab` and `slotsndtab` arrays, is compacted to remove empty slots and to form the final `vertadjtab` and `vertdlftab` arrays to be used for dichotomy search.

The data structures that are used during the computation of vertex global index update arrays are the following:

- `vertadjtab` and `vertdlftab`: these two arrays have been presented above. They are created only for receiver processes, and will be filled concurrently. They are of size  $((\text{commmax} + 1) * \text{orgprocnbr})$ , because in case a process is a sender receiver, it has to use a first slot to record the vertices it will keep locally, plus `commmax` for outbound communications. During the second stage of the algorithm, for some slot `i`, `vertadjtab[i]` holds the start global index of the chunk of vertices that will be kept, sent or received, and `vertdlftab[i]` holds the number of vertices that will be sent or received.

During the third stage of the algorithm, all this data will be compacted, to remove empty slots. After this, `vertadjtab` will be an array of global indices used for dichotomy search in `dgraphFold()`, and `vertdlftab[i]` will hold the adjustment value to apply to vertices whose global indices are comprised between `vertadjtab[i]` and `vertadjtab[i+1]`.

- `slotsndtab`: this array only has cells for receiver-slide slots, hence a size of  $((\text{commmax} + 1) * \text{procfldnbr})$  items. During the second stage of the algorithm, it is filled so that, for any non-empty communication slot `i` in `vertadjtab` and `vertdlftab`, representing a receive operation, `slotsndtab[i]` is the slot index of the corresponding send operation. During the third stage of the algorithm, it is used to compute the accumulated vertex indices across processes.

Here are some examples of redistributions that are computed by the `dgraphFoldComm()` routine.

```

1 orgvertcnttab = { 20, 20, 20, 20, 20, 20, 20, 1908 }
2 partval = 1
3 vertglbmax = 1908
4 Proc [0] (SND) 20 -> 0 : { [4] <- 20 }
5 Proc [1] (SND) 20 -> 0 : { [5] <- 20 }
6 Proc [2] (SND) 20 -> 0 : { [6] <- 20 }
7 Proc [3] (SND) 20 -> 0 : { [6] <- 20 }
8 Proc [4] (RCV) 20 -> 512 : { [0] -> 20 }, { [7] -> 472 }
9 Proc [5] (RCV) 20 -> 512 : { [1] -> 20 }, { [7] -> 472 }
10 Proc [6] (RCV) 20 -> 512 : { [2] -> 20 }, { [7] -> 452 }, { [3] -> 20 }
11 Proc [7] (RSD) 1908 -> 512 : { [4] <- 472 }, { [5] <- 472 }, { [6] <- 452 }
12 commmax = 4
13 commsum = 14

```

We can see in the listing above that some interleaving took place on the first receiver (proc. 4) before the sender receiver (proc. 7) did its first communication towards it.

```

1 orgvertcnttab = { 0, 0, 0, 20, 40, 40, 40, 100 }
2 partval = 1
3 vertglbmax = 100
4 Proc [0] (SND) 0 -> 0 :
5 Proc [1] (SND) 0 -> 0 :
6 Proc [2] (SND) 0 -> 0 :
7 Proc [3] (SND) 20 -> 0 : { [4] <- 20 }
8 Proc [4] (RCV) 40 -> 60 : { [3] -> 20 }
9 Proc [5] (RCV) 40 -> 60 : { [7] -> 20 }
10 Proc [6] (RCV) 40 -> 60 : { [7] -> 20 }
11 Proc [7] (RSD) 100 -> 60 : { [5] <- 20 }, { [6] <- 20 }
12 commmax = 4
13 commsum = 6

```

In the latter case, one can see that the pure sender that has been interleaved (proc. 3) sufficed to fill-in the first receiver (proc. 4), so the first communication of the sender receiver (proc. 7) was towards the next receiver (proc. 5).