

Scotch 7.0 Maintainer's Guide

(version 7.0.18)

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

October 31, 2026

Abstract

This document describes some internals of the libScotch library.

Contents

1 Introduction **3**

2 Coding style **3**
- 2.1 Typing . . . 4
  - 2.1.1 Spacing . . . 4
  - 2.1.2 Aligning . . . 4
  - 2.1.3 The `goto` keyword . . . 4
  - 2.1.4 The `return` keyword . . . 5
- 2.2 Indenting . . . 5
  - 2.2.1 Indenting and labels . . . 6
- 2.3 Comments . . . 6

3 Naming conventions **7**
- 3.1 File inclusion markers . . . 7
- 3.2 Variables and fields . . . 7
- 3.3 Functions . . . 9
- 3.4 Array index basing . . . 10

4 Structure of the libScotch library **10**

5 Files and data structures **11**
- 5.1 Decomposition-defined architecture files . . . 11

6 Data structure explanations **12**
- 6.1 `Dgraph` . . . 12
  - 6.1.1 Compactness . . . 12
- 6.2 `Dorder` . . . 13
  - 6.2.1 `DorderIndex` . . . 14
  - 6.2.2 `DorderLink` . . . 15
  - 6.2.3 `DorderNode` . . . 15
  - 6.2.4 `DorderCblk` . . . 16
- 6.3 `Graph` . . . 17
- 6.4 `Hgraph` . . . 20
- 6.5 `Kgraph` . . . 21
  - 6.5.1 Mappings . . . 22
- 6.6 `Mapping` . . . 22
- 6.7 `Order` . . . 24
  - 6.7.1 `OrderCblk` . . . 26
- 6.8 Hash tables . . . 27
  - 6.8.1 Data structure . . . 27
  - 6.8.2 Operation . . . 28
  - 6.8.3 Resizing . . . 29

7 Code explanations **31**
- 7.1 `archDecoArchBuild()` . . . 31
  - 7.1.1 Computing recursively the bipartitions . . . 32
  - 7.1.2 Computing the terminal distance matrix . . . 32
- 7.2 `dgraphCoarsenBuild()` . . . 33
  - 7.2.1 Creating the fine-to-coarse vertex array . . . 33
- 7.3 `dgraphFold()` and `dgraphFoldDup()` . . . 34

7.3.1 dgraphFold2() . . . 35
7.3.2 dgraphFoldComm() . . . 37
7.4 dmeshDgraphDual() . . . 40
7.4.1 Determining the node vertex range . . . 40
7.4.2 Creating node adjacencies . . . 41
7.4.3 Making node adjacencies available to concerned elements . . . 42
7.4.4 Creating the element-to-element adjacencies . . . 43
7.5 dorderCheck() . . . 43
7.5.1 Checking management data . . . 44
7.5.2 Checking permutations . . . 44
7.6 dorderTreeDist() . . . 45
7.6.1 Gathering locally-rooted column block data . . . 45
7.6.2 Sorting column block data . . . 45
7.6.3 Reconciling column block data . . . 46
7.6.4 Filling-in of user arrays . . . 46

8 Procedures for new developments and release 46
8.1 Adding methods to the libScotch library . . . 46
8.1.1 What to add . . . 46
8.1.2 Where to add . . . 47
8.1.3 Declaring the new method to the parser . . . 48
8.1.4 Adding the new method to the Make compilation environment 48
8.1.5 Adding the new method to the CMake compilation environment 49
8.2 Adding routines to the public API . . . 49
8.3 Release procedure . . . 49
8.3.1 Removal of debugging flags . . . 49
8.3.2 Symbol renaming . . . 50
8.3.3 Update of copyright year . . . 50
8.3.4 Update of version number . . . 50
8.3.5 Generation of documentation . . . 50
8.3.6 Creation of the local tag . . . 51
8.3.7 Merging . . . 51
8.3.8 Creation of the public tag . . . 51
8.3.9 Generation of the asset . . . 51

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:

```
1  for (hashbas = hashold - 1; hashtab[hashbas].vertnum != ~0; hashbas --) ;
```

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 The `goto` keyword

The use of the `goto` keyword should be restricted to the very specific cases when doing without it would significantly weigh down the coding style (e.g., through duplication of a long or complex piece of code), and/or would complicate the understanding of the code, and/or would impede performance (e.g., by adding a dedicated flag that would require extra tests within nested loops).

In the following code snippet, the use of a `goto` statement allows one to jump directly to a long block that would else have to be duplicated, or the execution of which would require to add a dedicated flag to pass in sequence the two tests that guard the block.

```
1         if (hashtab[helmend].vertnum != velmnum) { /* If edge not yet
              created */
2           ... /* Record edge in hash table */
3           nghbnbr = ...; /* Set number of remaining neighbors */
4           goto test; /* Is it enough to create the edge ? */
5         }
6         if (hashtab[helmend].vertend == velmend) { /* If hash slot found */
7           if (... > 0) { /* If edge not already created */
8             hashtab[helmend].nghbnbr = -- nghbnbr; /* One more instance */
9 test: if (nghbnbr <= 0) { /* If threshold reached */
10            ...
```

Label names for `goto` statements should be in lowercase only, as their visibility is limited to the inside of a routine, like local variables. They should be short and telling, e.g. : "`fail`", "`loop`", "`nomatch`", "`redo`", etc. Their indentation rules are explained below.

Labels should be short and standardized. Typical label names comprise:

- `fail`: abort some computation because of failure.
- `loop`: skip some computation and start next iteration of some loop from the beginning.
- `redo`: restart a process because some condition has changed.
- `skip`: skip some computation (e.g., to avoid or exit some loop or function).

When several labels of the same kind are present in the same function, they are numbered consecutively with an integer suffix, e.g.: "`fail1`", "`fail2`". Labels are numbered by order of definition in the code, and not by of first occurrence of use.

2.1.4 The `return` keyword

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.

2.2.1 Indenting and labels

Label names for `goto` statements always start at column zero, and the colon should immediately follow the label. If at least one blank character can be placed between the colon and the indented text, the label should be placed on the same line as the text; else, the label should be placed on a separate line, immediately followed by the next line, without any blank line in-between. If a blank line is deemed necessary for the sake of readability, it should be placed before the label.

```
    goto fail;
    ...
  }

fail:
  ...
```

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 colums 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[1], 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).
- `vlbl`: user-defined vertex label (at the user API level).
- `edge`: edge (*i.e.*., arcs, in fact).

[1] See e.g. `https://gcc.gnu.org/onlinedocs/cpp/Implementation-limits.html`

- `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. $0 \leq$ **vertnum** $<$ **vertnbr** if the vertex index is un-based, and **baseval** $\leq$ **vertnum** $<$ **vertnnd** if the index is based, that it, 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` = `xxxxnbr` + `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 explicitely 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 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.

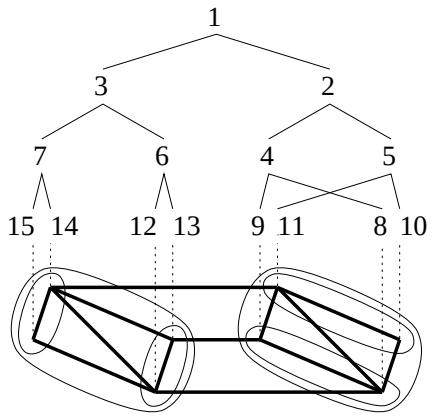
Also, while the "`deco 0`" format lists only the characteristics of terminal domains (in terms of weights and labels), the "`deco 1`" format provides these for all domains, so as to speed-up the retrieval of the size, weight and label of any domain, whether it is terminal or not.

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.




```
deco 0
8 15
0 1 15
1 1 14
2 1 13
3 1 11
4 1 12
5 1 9
6 1 8
7 1 10
1
2 1
2 1 2
1 1 1 2
3 2 1 1 2
2 2 2 1 1 1
3 2 3 1 2 2 1
```

Figure 1: "`deco 0`" target decomposition file for UB(2, 3). The terminal domain numbers associated with every processor define a unique recursive bipartitioning of the target graph. In the file format, the list of terminal domains, with their label, weight, and terminal domain number, is followed by the triangular distance matrix between all terminal domain numbers.

6 Data structure explanations

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

6.1 `Dgraph`

6.1.1 Compactness

Like centralized `Graph`s, distributed graphs can be compact, that is, have their `vendloctab` array merged with their `vertloctab` array. In this case, on all processing elements, `vendloctax` = `vertloctax` + 1. However, there may be situations when this expression holds on at least one processing element, while the distributed graph is not compact. It is for instance the case when a processing element holds

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

```
2 2 2 2 1 2 2 1
3 1 2 2 1 2 2 2
3 1 2 2 1 2 1 2
1 1 2 2 3 2 3 1
2 2 3 1 3 2 3 2
1 3 3 1 2 2 1 2
1 1 2 2 1 1 1 2
2 1 2 2 1 1 1 2
2 2 3 3 2 2 3 1
2 2 1 3 2 1 2 2
1 2 2 1 1 2 2 2
1 1 1 3 3 2 3 3
2 1 2 3 3 2 1 2
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.

only one local vertex, and the `vendloctab` array is allocated just after the `vert loctab` array, in the same memory group. In order to eliminate such an ambiguity, the `flagval` field of the `Dgraph` structure possesses a `DGRAPHHASVENDLOC` flag which, when set, indicates that the distributed flag is not compact. Only in the case when this flag is not set can it be considered that the distributed graph is compact.

6.2 `Dorder`

Distributed orderings are data structures used in PT-Scotch to represent orderings distributed on a set of processing elements. Like for the centralized ordering of type `Order` (see Section 6.7), a distributed ordering consists of an inverse permutation, which provides the old indices of the reordered vertices, and a column block decomposition of the reordered matrix, to help perform more efficient block computations at the solve stage. The column block decomposition is defined as a tree structure, the nodes of which, of type `DorderCblk`, represent column blocks tree nodes containing consecutive, reordered vertices. A tree node may have children nodes, which represent the decomposition of a column block into sub-column blocks, e.g., when a subdomain is decomposed into two separated subdomains and a separator.

Because its column blocks are distributed across multiple processing elements, the `Dorder` data type is much more complex than the `Order` data type. Hence, it is important to fully understand the `Order` data type before delving into the meanders of the `Dorder` data type and its ancillary data types. The main difference between the two is that, since graphs are distributed across multiple processing elements, column block information has to be duplicated on all the processing elements which contain a piece of a given graph. In order to reconciliate this information, to provide a centralized block column ordering, all distributed column block tree node structures are identified by a `DorderIndex` data structure.

The distributed column block tree data structure, created by way of parallel graph separation algorithms, always ends up in leaves, when nested dissection no

longer succeeds or when the distributed subgraphs are folded onto single processing elements. As soon as the latter happens, a purely sequential graph ordering process can take place on each of them. This leads to the creation of a leaf `Dorder Cblk` node, into which the resulting locally-computed, centralized column block sub-tree is compacted as an array of `DorderNode` data structures. From the above, at the time being, the distributed column block tree structure contains either nested dissection nodes, of type `DORDERCBLKNEDI`, leaf nodes, of type `DORDER CBLKLEAF`, or disconnected components nodes of type `DORDERCBLKDICO`. In order to facilitate the integration of centralized column block sub-trees into a global distributed column block tree, the values of the type flags are the same for the `Dorder` and `Order` data types.

The fields of the `Dorder` data structure are the following:

`baseval`
: Base value for the inverse permutation.

`vnodglbnbr`
: Overall number of node vertices to order across all processing elements. For graph orderings, this number is equal to the number of non-halo vertices in the initial graph.

`cblklocnbr`
: Local number of locally-rooted column blocks. This number is the sum of the number of centralized column blocks, of type `DorderNode`, held by the current processing element, plus the number of distributed column block tree nodes, of type `DorderCblk`, the `proclocnum` index of which is equal to the rank of the processing element. This allows one to count only once each distributed column block tree node, when summing the `cblklocnbr` fields over all processing elements.

`linkdat`
: Start of the doubly-linked list of distributed column block tree nodes, of type `DorderCblk`, on the given processing element. This list is circular, to allow for the insertion of new nodes at the end of the list in constant time.

`proccomm`
: MPI communicator for managing the distributed ordering. It should be the same as that of the initial distributed graph to be ordered.

`proclocnum`
: Rank of the given processing element within the communicator.

`mutelocdat`
: When multi-threading is activated, allows one to create critical sections to update the ordering data in a thread-safe manner.

6.2.1 `DorderIndex`

Since the ordering data structure is distributed, pointers cannot be used to refer to parent or children column block tree node data structures across processing elements. The `DorderIndex` data type defines an identifier for column block tree nodes. These identifiers are unique, in the sense that, on each processing element, no two `DorderCblk` structures will have the same identifier. However, several `Dorder Cblk` structures may bear the same `DorderIndex` values on different processing

elements, in the case when they are siblings which maintain the local information about the same distributed column block tree node.

The fields of the `DorderIndex` data type are the following:

`proclocnum`
: Smallest rank among the processing elements on which a copy of the column block tree node resides.

`cblklocnum`
: Local number of the column block tree node data structure on the processing element of aforementioned rank.

6.2.2 `DorderLink`

Since distributed column block tree nodes, of type `DorderCblk`, are created on the fly on each processing element, are in small numbers, and are heavy structures, they are not stored in a single resizable array, but as individual cells which are allocated when needed. Consequently, these structures have to be linked together, for proper management.

The `DorderLink` data type aims at chaining all `DorderCblk` structures in a circular, doubly-linked list. New nodes are inserted at the end of the list, such that a simple traversal yields nodes in ascending creation order, which is essential for locally-rooted nodes when gathering them to create a centralized ordering. The `DorderLink` structure is the first field of the `DorderCblk` structure, so that a simple pointer cast allows one to retrieve the tree node structure from the current link.

The fields of the `DorderLink` data structure are the following:

`nextptr`
: Pointer to the next distributed column block tree node created on the given processing element.

`prevptr`
: Pointer to the previous distributed column block tree node created on the given processing element.

6.2.3 `DorderNode`

The distributed column block tree data structure ends up in leaves, when either the parallel nested dissection stops, or when distributed subgraphs are located on single processing elements. In the first case, the distributed subgraph is centralized, after which, in both cases, a centralized ordering strategy is applied to the centralized subgraph, and a centralized block ordering is computed. This centralized block ordering is represented as an `Order` data structure, containing an inverse permutation and a tree of `OrderCblk` nodes. Since the distributed ordering will eventually have to be centralized, the local, centralized orderings will have to be compacted and sent to the root processing element. In order to anticipate this and to save space, once a centralized ordering is computed on some processing element, the resulting column block tree is compacted into a single array of `Order Node` cells.

The fields of the `DorderNode` data type are the following:

`fathnum`
: Un-based index of the father node of the given node in the node array, or -1

if the given node is a local root and has to be connected to the father of the local leaf of the distributed column block tree.

`typeval`
: Type of centralized column block tree node. The admissible values are constants of the kind `ORDERCBLK*`.

`vnodnbr`
: Number of node vertices in the column block.

`cblknum`
: Rank of the tree node among the children of its father, starting from zero.

Like for the `DorderCblk` data type, there are no references from a node to its children, but a reference from each node to its father, with all information needed to rebuild a global centralized column block tree when all node information is centralized on a single processing element.

6.2.4 `DorderCblk`

The `DorderCblk` data type represents distributed column block tree nodes within distributed orderings. A tree node may be a leaf node, or have children nodes which describe the decomposition of a column block into sub-column blocks, e.g., when a graph is decomposed into two separated subgraphs and a separator.

Since, by nature, every distributed column block tree node concerns a set of vertices distributed across a set of processing elements, each of the latter holds a copy of the tree node, the identifier of which, of type `DorderIndex`, contains identical information: the smallest rank among the involved processing elements within the communicator used to manage the distributed ordering, and an index incrementally generated on this processing element. Unlike for the `Order` data type, there are no pointers from a tree node to its child nodes; on the opposite, the `DorderCblk` node contains a `DorderIndex` referring to its father node. The only information a tree node will hold about its children is their number.

The fields of the `DorderCblk` data type are the following:

`linkdat`
: Doubly-linked list structure to chain together all the `DorderCblk` structures on a given processing element.

`ordelocptr`
: Pointer to the distributed ordering to which the given distributed column block tree node belongs.

`typeval`
: Type of tree node; at the time being, it is either `DORDERCBLKNEDI` for a nested dissection node, `DORDERCBLKLEAF` for a leaf node, or `DORDERCBLK DICO` for disconnected components.

`fathnum`
: Identifier of the father of the given column block tree node. If the given tree node is a root, the value of the father index is `{ 0, -1 }`.

`cblknum`
: Identifier of the given column block tree node. The process number is the smallest rank among all the processing elements sharing node vertices, and the local number is provided incrementally on this processing element.

`ordeglbval`
: Un-based global start index of the node vertices in the distributed column block tree node.

`vnodglbnbr`
: Number of node vertices contained in the distributed column block tree node, over all the involved processing elements. If the column block has sub-column blocks, the sum of all the `vnodglbnbr` values of the sub-column blocks must be equal to the `vnodglbnbr` of the column block.

`cblkfthnum`
: Index of the given column block tree node among its siblings, starting from zero.

`data`
: Union field holding the information concerning either the leaf node or the nested dissection node. This field has two sub-fields:

  `leaf`
  : Leaf field, which has the following sub-fields:

    `ordelocval`
    : Un-based start index in the global inverse permutation array for the local vertices.

    `vnodlocnbr`
    : Number of node vertices in the given permutation fragment.

    `periloctab`
    : Pointer to the local, un-based, inverse permutation fragment array, of size `vnodlocnbr`. The values of the `periloctab` array are based according to the `baseval` field of the `Dorder` data type.

    `nodelocnbr`
    : Number of local column block tree nodes associated with the permutation fragment.

    `nodeloctab`
    : Pointer to the local, un-based, array of local column block tree nodes, of size `nodelocnbr`.

    `cblklocnum`
    : Un-based index, in `nodeloctab`, of the root local column block tree node.

  `nedi`
  : Nested dissection field. This field has a single sub-field:

    `cblkglbnbr`
    : Number of sub-column blocks within this column block. For nested dissection, this number is either 2 (two separated parts and no separator) or 3 (two separated parts and a separator).

6.3 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 baseval $\leq i <$ (vertnbr + baseval), (vendtax[i] − 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.

vnumtax
: When the current graph is a subgraph of some initial graph, this based array, of size vertnbr, holds the initial vertex indices of the subgraph vertices. This array is not defined (*i.e.*, vnumtax = NULL) when the graph is the initial graph.

edgetax
: Based array, of a size equal at least to ($\max_i$(vendtax[i]) − baseval), holding the adjacency array of every vertex.

edlotax
: Optional based array, of a size equal at least to ($\max_i$(vendtax[i]) − 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

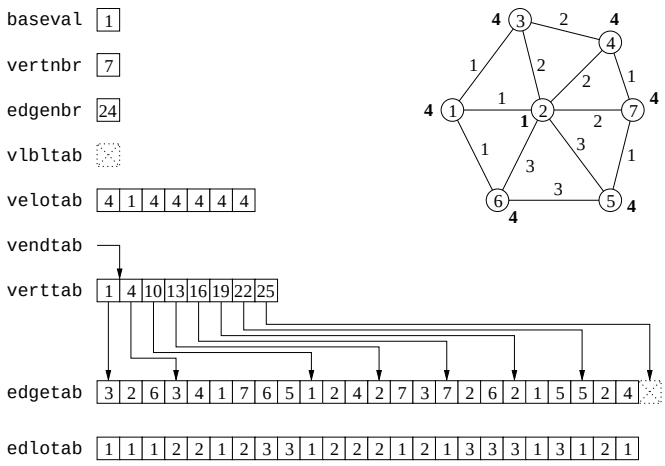


Figure 3: Sample graph and its description using a compact edge array. Numbers within vertices are vertex indices, bold numbers close to vertices are vertex loads, and numbers close to edges are edge loads. Since the edge array is compact, `verttax` is of size `vertnbr` + 1 and `vendtax` points to `verttax`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

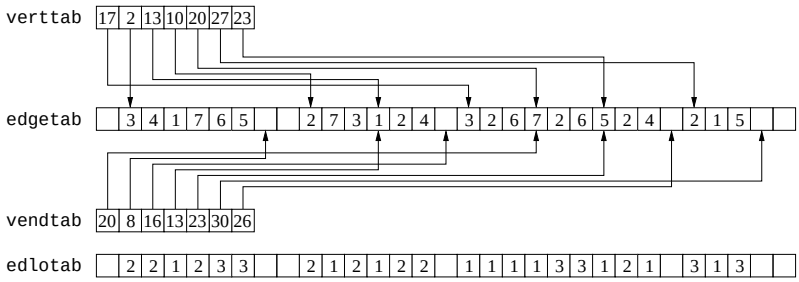


Figure 4: Adjacency structure of the sample graph of Figure 3 with disjoint edge and edge load arrays. Both `verttax` and `vendtax` are of size `vertnbr`. This allows for the handling of dynamic graphs, the structure of which can evolve with time.

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 is. 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, 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.

6.4 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, $0 \leq$ `vnohnbr` $\leq$ `s.vertnbr`.

`vnhdtax`
Array of after-last indices in `s.edgetax` of non-halo vertex adjacency sub-

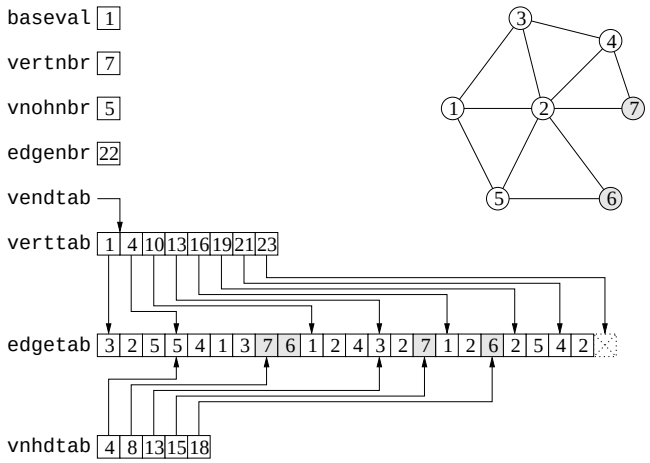


Figure 5: Sample halo graph and its description using a compact edge array. Numbers within vertices are vertex indices. Greyed values are indices of halo vertices. Halo vertices have the highest indices in the graph, and are placed last in the adjacency sub-arrays of each non-halo vertex.

arrays. Since this information only concerns non-halo vertices, `vnhdtax` is of size `vnohnbr`, not `vertnbr`. For any non-halo vertex i, with `baseval` $\leq i <$ (`vnohnbr` + `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, $0 \leq$ `vnlosum` $\leq$ `s.velosum`.

`enohnbr`
: Number of non-halo arcs in graph. Hence, $0 \leq$ `enohnbr` $\leq$ `s.edgenbr`.

6.5 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 redundency is the price to pay for lighter memory management.

6.5.1 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 Kgraphs (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 Kgraphs. It is protected from double memory freeing by not setting the MAPPINGFREEDOMN flag in field r.m.flagval.

6.6 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 biparitioning 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 indicating 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, $1 \leq$ `domnnbr` $<$ `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.

6.7 `Order`

Orderings are data structures used in SCOTCH to represent fill-minimizing block orderings of adjacency matrices represented as graphs. A block ordering, contained in the `Order` data structure, is defined by an inverse permutation, which provides the old indices of the reordered vertices, and a column block decomposition of the reordered matrix, to help performing more efficient block computations at the solving stage.

Inverse permutations are used, instead of direct permutations, because their processing is more local: when ordering some subgraph, the only ordering information to provide is the un-based start index, usually called `ordenum`, in the inverse permutation vector, usually called `peritab`, while the `vnumtax` array of the `Graph` structure holds the values of the vertex indices to write in the sub-array of `peritab` starting at index `ordenum`, of a size equal to the number of concerned vertices in the `Graph`. Once an ordering is computed, it is straightforward to compute the direct permutation `permtab` from the inverse permutation `peritab`, in case it is needed.

The column block decomposition is defined as a tree structure, whose nodes, of type `OrderCblk`, represent column blocks containing consecutive, reordered vertices. A tree node may have ordered children nodes, which represent the decomposition of a column block into sub-column blocks, e.g., when a subdomain is decomposed into two separated subdomains and a separator.

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

`flagval`
: Flag that indicates whether the `peritab` inverse permutation array has to be freed on exit.

`baseval`
: Base value for inverse permutation values.

`vnodnbr`
: Number of vertex nodes in the ordering. When the associated graph structure is a `Graph`, this number is equal to its `vertnbr` field; when it is a `Mesh`, it is equal to its `vnodnbr` field.

`treenbr`
: Number of tree nodes in the ordering. This number is equal to 1 when only the root tree node is present, and is incremented each time a new tree node is added to the tree structure.

`cblknbr`
: Number of column blocks in the ordering. This number is equal to 1 when only the root tree node is present. When some column block is decomposed into c sub-column blocks, it is increased by $(c - 1)$, since this represents the number of additional column blocks in the structure.

`rootdat`
: Root column block of the ordering. This structure, of type `OrderCblk`, is initialized to contain all `Graph` vertices, or `Mesh` vertex nodes, in a single column block, after which reordering algorithms are applied and lead to the creation of sub-column blocks, e.g., in the case of nested dissection.

`peritab`
: Pointer to the inverse permutation array.

`mutedat`
: Mutual exclusion lock. When multi-threading is activated, it allows to create critical sections to update the ordering data in a thread-safe manner.

6.7.1 `OrderCblk`

Column blocks are sets of reordered unknowns which are likely to be processed efficiently together when solving the linear system, e.g., using BLAS block computation routines. The column block decomposition of the reordered matrix is represented as a tree whose nodes are instances of the `OrderCblk` data structure. The column block decomposition tree will be used to create the block elimination tree of the unknowns of the linear system, which amounts to linking each column block to a father block. This building is performed by the `orderTree()` routine.

A column block tree node `OrderCblk` is defined by its type (*i.e.*, whether it is a leaf, a nested dissection node, etc.), its width (*i.e.*, the number of node vertices it contains), and, if it is not a leaf, the description of the sub-column blocks it contains. The main fields of the `OrderCblk` data structure are the following:

`typeval`
: Set of flags that define the nature of the column block tree node. They must be the same as the distributed column block tree node flags of the `Dorder Cblk` distributed column block data structure. Consequently, these flags must be separate bits, so that values can be or-ed (especially, concerning `ORDER CBLKLEAF` in `hdgraphOrderNd()`). These flags are the following:

  `ORDERCBLKLEAF`
  : Leaf column block (before it is subdivided into sub-column blocks, or definitely). In this case, the other fields of the column block tree node are such that `cblknbr` = 0 and `cblktab` = `NULL`.

  `ORDERCBLKNEDI`
  : Nested-dissection separator tree node. The separator is always the last sub-column block. Hence, if the separator is not empty, the node has three sub-column blocks (hence `cblknbr` = 3), while, if the separator is empty, the column block tree node has only two sub-column blocks (hence `cblknbr` = 2). None of the separated parts can be empty (else, the tree node would be of type `ORDERCBLKSEQU`).

  `ORDERCBLKDICO`
  : Disconnected components tree node. It contains an arbitrary number (always strictly greater than 1) of sub-column blocks, which represent disconnected components to be ordered independently. Since the sub-column blocks are not connected, their father in the elimination tree will not be the column block tree node itself, but its father (or none if the column block is the root column block, *i.e.*, the `rootdat` field of the `Order` data structure).

  `ORDERCBLKSEQU`
  : Sequential tree node. It contains an arbitrary number (always strictly greater than 1) of sub-column blocks, which represent mutually dependent blocks. Consequently, the father of each sub-column block in the elimination tree will be the next sub-column block, except for the last sub-column block, whose father will be the column block itself.

`vnodnbr`
: Number of nodes (*i.e.*, vertices) contained in the column block. If the column block has sub-column blocks, the sum of all the `vnodnbr` values of the sub-column blocks must be equal to the `vnodnbr` of the column block.

`cblknbr`
: Number of sub-column blocks. If the column block is a leaf, `cblknbr` = 0 and `cblktab` = `NULL`.

`cblktab`
: Array of `cblknbr` structures of type `DorderCblk`, which hold the data about the sub-column blocks if the column block is not a leaf. `cblktab` has to be freed on exit.

6.8 Hash tables

Hash tables are used quite often in SCOTCH. However, since their use is problem-dependent, and the code that implements them is small, no generic data structure has been created to handle them. Each instance is created *ad hoc*, at the expense of a slight duplication of code. This provides better readability, as macros would hide their meaning.

These hash tables use *open addressing* and *linear lookup*. Because of open addressing, removal of individual items is not possible, which is not a concern for our use cases. Since linear lookup can prove very expensive when tables get full, implementations in SCOTCH make sure that tables are not filled-in at more than 25% capacity. Our experiments showed that this maximum load factor guarantees that simple collisions happen in only about 1% of the cases, and that more-than-double collisions are almost nonexistent. When the maximum number of items cannot be known, resizing is implemented, to enforce this maximum load factor.

A typical use of these hash tables is to process the (common) neighbors of several vertices. For instance, when coarsening a graph, the adjacencies of two mated vertices have to be merged into a single adjacency, taking care of duplicate edges. In this case, edge merging must be performed when one of the vertices is connected to two neighbors that will be merged together, or when the two vertices are connected to the same neighbor vertex or to two vertex neighbors to be merged together. The edge connecting the two mated vertices must also be removed, if it exists.

6.8.1 Data structure

The underlying data structure of a SCOTCH hash table is an array of a size which is always a power of two. This constraint aims at providing cheap ways for array index bounding, by turning expensive integer modulus operations into cheap bitwise 'and' operations using a dedicated bit mask variable.

In each use case, the semantics of cell data must allow one to indicate unambiguously whether a cell is empty or full. Since hash indices are most often vertex or edge indices, and these indices are always positive or null (*i.e.*, `baseval` is always positive or null), '−1' is commonly used as a marker value in cell fields to indicate an empty cell. To fill-in the hash table array at initialization time with '−1' values in one sweep, one can use a `memset()` routine with the '`~0`' byte value (all 1's), assuming negative integer numbers are coded in two's complement.

When a hash table is used repeatedly (e.g., for all vertices of a graph), re-initializing the whole hash table memory area when only a few of its cells have been touched may prove expensive. The solution is to embed in each cell a pass number (e.g., the vertex number), so that a cell is considered empty when its pass number is not equal to the number of the current pass. This is consistent with initializing the table array with '−1's, since pass numbers will always be greater than this value.

Adding this extra field increases the size of each cell, hence of the whole array, but reduces the overall number of memory writes while preserving cache locality.

For instance, in the case of graph coarsening discussed above, a hash table cell may have the structure described below: `vertnum` is the pass number (*i.e.*,the index of the current coarse vertex the adjacency of which is being built), `vertend` is the hash index (the number of the coarse vertex neighbor for which an edge must be maintained), and `edgenum` is the index of the edge to be created in the coarse graph edge array.

```
typedef struct HashCell_ {
 Gnum vertnum; /*+ Origin vertex (i.e. pass) number +*/
 Gnum vertend; /*+ Other end vertex number +*/
 Gnum edgenum; /*+ Number of corresponding edge +*/
} HashCell;
```

6.8.2 Operation

As said, the size of the hash table array must be a power-of-two size at least greater than four times the expected maximum number of items, and its contents must be initialized with '−1' values. All of this can be performed with the following code.

```
 itemmax = ...; /* Plausible maximum number of items */
 for (hashmax = 2; hashmax > itemmax; hashmax *= 2) ; /* Find upper power
     of two */
 hashsiz = hashmax * 4; /* Array size guarantees 25% load factor */
 hashmsk = hashsiz - 1; /* Bit mask is range of 1's */

 hashtab = (Hash *) memAlloc (hashsiz * sizeof (HashCell)); /* Allocate
     hash table array */
 memSet (hashtab, ~0, hashsiz * sizeof (HashCell)); /* Fill-in array with
     '-1' values */
```

In the above, `hashtab` is the pointer to the hash table array, `itemmax` is the maximum number of items, `hashmax` is the maximum number of elements to be inserted in the hash table, `hashsiz` is the size of the array (in number of cells), and `hashmsk` is the bit mask for index bounding. In a real code, not all of these variables are explicitly named and computed, because there exist simple relationships between them. For instance, with respect to `hashmsk`, which is always present because it must be easily available in hash loops, we have $\texttt{hashsiz} = \texttt{hashmsk} + 1$ and $\texttt{hashmax} = (\texttt{hashmsk} - 3)/4$. Multiplications and divisions by powers of two can be cheaply performed by way of bit shift operators (*i.e.*, '<<' and '>>'). Hence, $\texttt{hashmax} = \texttt{hashmsk} >> 2$, which also discards the unwanted low-order bits.

Once the hash table is set-up, it can be used efficiently with only a few lines of code. To prevent data clustering in the array, the initial index is computed by way of a multiplication by a number which is prime with the size (hence, an odd number). For instance, from a vertex number `vertnum`: $\texttt{hashnum} = \texttt{vertnum} \times$ HASHPRIME. Typically, the prime number is 17 or 31, that is, a number such that multiplying by it amounts to a one-bit shift and an addition/subtraction only (e.g., $v \times 17 = v << 4 + v$ and $v \times 31 = v << 5 - v$).

As said, lookup is linear: once a hash index `hashnum` is computed, if the corresponding cell is already full, the next index will be computed by incrementing `hashnum` and performing a modulus operation using the bit mask, as $(\texttt{hashnum} + 1)\%\texttt{hashsiz} = (\texttt{hashnum} + 1)\&\texttt{hashmsk}$.

In the case of graph coarsening taken as an example, the hash table can be used in the following way.

```
1  for (coarvertnum = ...) { /* For each coarse vertex to consider */
2    for (finevertnum = ...) { /* Enumerate fine vertices to merge in it */
3      for (fineedgenum = fineverttax[finevertnum]; /* For all fine edges of
           current fine vertex */
4          fineedgenum < finevendtax[finevertnum]; fineedgenum ++) {
5        coarvertend = finecoartax[fineedgetax[fineedgenum]]; /* Get coarse
             number of end fine vertex */
6
7        if (coarvertend != coarvertnum) { /* If not end of collapsed edge */
8          for (hashnum = (coarvertend * HASHPRIME) & hashmsk; ;
9              hashnum = (hashnum + 1) & coarhashmsk) { /* For all possible
                   hash slots */
10           if (coarhashtab[hashnum].vertnum != coarvertnum) { /* If slot is
                 empty */
11             hashtab[hashnum].vertnum = coarvertnum; /* Create hash slot in
                   table */
12             hashtab[hashnum].vertend = coarvertend;
13             hashtab[hashnum].edgenum = ...; /* Set edge number in coarse
                   graph */
14             break; /* Give up hashing as it succeeded */
15           }
16           if (hashtab[hashnum].vertend == coarvertend) { /* If coarse edge
                 already exists */
17             ... /* Manage merging of fine edge in coarse one */
18             break; /* Give up hashing as it succeeded */
19           }
20         }
21       }
22     } /* Go on searching for an empty cell */
23   }
24 }
```

6.8.3 Resizing

When the number of items inserted in the table becomes greater than `hashmax` (*i.e.*, above the 25% load factor), resizing takes place. This may be a complex procedure, depending on the semantics of the data types (e.g., when hash slot indices are referenced in other data structures which must then also be updated). However, the process is quite straightforward.

First, the size of the array is doubled, by way of a `realloc()` call, and the second half of the enlarged array is initialized, by way of a `memset(,~0,)` call. Then, cell locations must be updated, according to their new hash indices and the linear lookup policy.

This process is performed in two phases. The first phase concerns the block of all the non-empty cells contiguous to the end of the old hash table (*i.e.*, the end of the first half of the resized hash table). If the last cell of the old table is empty, this last block does not exist. If it does, then all its cells are processed in ascending order: their new hash index is computed and, if it differs from the current one, the cell is moved to its new slot, possibly in the new half of the array. Then, in the second phase, all the cells from the beginning of the array to the last cell before those possibly processed in the first phase are processed similarly, also in ascending order. Together, these two segments cover all the cells of the old hash table.

This two-phase approach is necessary to allow for moving all the cells of the hash array without causing bugs. Assume the old hash table contains only two cells: a first cell at the very end of the array, placed here because its hash index is indeed the last index of the array, and a second cell at the very beginning of the array, which was placed here because its hash index was also the index of the last cell but

has been subsequently set to zero in the lookup phase since the last cell was already full. If only the second phase would take place, the first cell of the array would have its index recomputed first, and may still be the end of the first half of the new array, but, since this slot is still busy, the cell would be moved just after it, that is, now that the table is resized, at the very beginning of the second half of the resized array, which is empty. Then, the last cell would have its index recomputed and, if its new index would differ from its former one, because of the different modulus, it would be moved elsewhere. Hence, the first cell would no longer be accessible.

These two phases can be combined into the same factored code, by way of the outer loop exemplified below.

```
1  hashtab = memRealloc (hashtab, 2 * hashold * sizeof (HashCell)); /*
       Resize hash table */
2  memSet (hashtab + hashold, ~0, hashold * sizeof (HashCell)); /*
       Initialize second half */
3
4  for (hashbas = hashold - 1; hashtab[hashbas].vertnum == vertnum; hashbas
       --) ; /* Find start index of last block */
5  hashnnd = hashold; /* First segment to reconsider ends at the end of the
       old array */
6
7  while (hashnnd != hashbas) { /* For each of the two segments to consider
       */
8    for (hashnum = hashbas; hashnum < hashnnd; hashnum ++) { /* Re-compute
         position in new table */
9      if (hashtab[hashnum].vertnum == vertnum) { /* If hash slot used in
           this pass */
10       vertend = hashtab[hashnum].vertend; /* Get hash key value */
11       for (hashnew = (vertend * HASHPRIME) & hashmsk; ; hashnew = (hashnew
             + 1) & hashmsk) {
12         if (hashnew == hashnum) /* If hash slot is the same */
13           break; /* There is nothing to do */
14         if (hashtab[hashnew].vertnum != vertnum) { /* If new slot is empty
               */
15           hashtab[hashnew] = hashtab[hashnum]; /* Copy data to new slot */
16           hashtab[hashnum].vertnum = ~0; /* Mark old slot as empty */
17           break;
18         }
19       } /* Go on searching */
20     }
21   }
22
23   hashnnd = hashbas; /* End of second segment to consider is start of
         first one */
24   hashbas = 0; /* Start of second segment is beginning of array */
25 } /* After second segment, hashbas = hashnnd = 0 and loop stops */
```

In the above, `hashold` is the size of the old hash table, `hashbas` is the start index of the current segment, and `hashnnd` is the end index of the current segment. For the first segment, the loop always starts on an empty cell; this allows one to handle smoothly the case when there is no last block. Since there always exists at least one empty cell in the array, because of the load factor, the loop on the first segment will never start off the bounds of the array. Like before, the `vertnum` field of hash cells is the current pass number, acting as an occupation flag, and `vertend`, the hash key value of the cell, is used to compute the new hash index. The emptiness test "`if (hashtab[hashnew].vertnum != vertnum)`" can also be written: "`if (hashtab[hashnew].vertnum == ~0)`", since the two cases in which an empty cell is found consist in the new hash value indexing either the new part of the hash table, or a cell that has been cleared after moving. In both cases, the value

of the pass number has been explicitly set to '~0'.

When resizing takes place within a search loop, it may happen that the location of the empty slot changes when the mask changes. Hence, writing data to some empty slot after resizing may result in data loss. To prevent such case, either resizing must happen before the hash table index is computed, or the computation must be restarted by jumping back to the beginning of the loop, as exemplified below.

```
1  redo:
2   for (hashnum = (vertnum * HASHPRIME) & hashmsk; ; hashnum = (hashnum + 1)
        & hashmsk) {
3     if (hashtab[hashnum].vertnum != vertnum) { /* If vertex not found */
4       if (hashnbr >= hashmax) { /* If hash table full, resize it */
5         hashResize (&hashtab, &hashsiz, &hashmax, &hashmsk);
6         goto redo; /* Recompute hashnum with new mask */
7       }
8
9       hashtab[hashnum].vertnum = vertnum; /* Record new vertex */
10      ...
11      hashnbr ++; /* One more item added to hash table */
12    }
13    ...
14  }
```

Depending on the use case, in each concerned Scotch routine, relevant variables and fields may have different names and semantics.

7 Code explanations

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

7.1 archDecoArchBuild()

The `archDecoArchBuild()` routine creates a decomposition-defined architecture of type "`0`", that is, a "`deco 0`" architecture, from a graph and an optional vertex list.

The "`deco 0`" decomposition-defined architecture is described, from the user's perspective, in the *Scotch User's Guide*. It builds a target architecture from a graph by relying on a set of recursive bipartitions of the graph to provide the successive target architecture subdomains as well as a lower triangular, diagonal-less, distance matrix of pre-computed distances between subdomains. The tree of recursive bipartitioning is materialized through the labeling of the graph vertices: the initial domain, comprising all wanted graph vertices, is numbered as 1, and the bipartition of some domain of number i yields two subdomains numbered $2i$ and $(2i + 1)$. These domain numbers, borne by the graph vertices (as "terminal" domains, that is, domains of size 1), allows one to reconstruct the sequence of bipartitions applied to the initial graph, as shown in Figure 1, page 12.

In practice, during mapping computations, one needs to know quickly the weights of all subdomains, and not only of the terminal ones, as well as the distances between all such subdomains. The enlarged domain list and distance matrices can be computed on the fly when loading a "`deco 0`" architecture, or can be pre-computed once for good and stored as a "`deco 1`" target architecture, described in Section 5.1 (the "`deco 2`" format is of a completely different nature).

7.1.1 Computing recursively the bipartitions

Recursive bipartitions are computed independently from each other. A job queue `joblink` chains all the bipartitioning jobs yet to be performed, in arbitrary order, while job data is stored in the `jobtab` array. The current state of the graph bipartitions is maintained in a `Mapping` structure `mappdat`. Since a mapping must be attached to a valid target architecture, a "`vcmplt`" variable-sized complete graph target architecture is created in `archdat`; its benefit is that the numbering scheme of the "`vcmplt`" target architecture is indeed the binary tree numbering scheme discussed above. Hence, the `mappdat` data structure is fully operational, and represents the current state of the recursive bipartitioning process.

The `archDecoArchBuild()` routine also takes a vertex list `tgtlisttab` as parameter, so that a target architecture can be created from a subset of vertices of the given graph. When this list is present, an induced graph is created from the initial graph `tgtgrafptr` in the initial job graph field. This has the current inconvenient of depriving the graph bipartitioning algorithm of the locality information between disconnected subparts, and will be changed in future versions.

Once the initial job is placed in the job list, the recursive bipartitioning process takes place. As long as the list is not empty, the first job in list is picked. The job graph data is cloned into the active bipartitioning `Bgraph` structure `actgrafdat`, which is completed by a call to `bgraphInit2()`. The graph is then bipartitioned by calling `bgraphBipartSt()`. If no partition is created (that is, all graph vertices are placed in the same part), an error is raised, because else the bipartitioning process would loop infinitely.

The result of each bipartitioning is accumulated in the `mappdat` structure. the `parttax` of this `Mapping` yields, for each vertex of the initial graph, onto which domain it is currently mapped. Once a bipartitioning has been computed, vertices in the first part point to the same domain, which is updated to refer to the first "`vcmplt`" subdomain, while the `parttax` array cells of the vertices of the second part are updated to point to a new domain, created at the current end of the part array, indexed by `mappdat.domnnbr`, and which is set to the second "`vcmplt`" subdomain.

After all jobs have been processed, the `mappdat` structure holds the complete mapping of the graph vertices (or just those belonging to `tgtlisttab`): the `parttax` cells index the cells of the `domntab` array, which contain the "`vcmplt`" terminal domain numbers. Graph vertices that do not belong to the vertex list keep their initial `partax` value of `0`, thus pointing to invalid mapping data; however, this information will never be used. When a vertex list is provided, an induced graph is created, which possesses a `vnumtab`; this index array is used to write vertex mapping data into the `parttax` array of `mappdat` at the proper initial vertex indices. Hence, domain information has always to be retrieved through the `parttax` array.

7.1.2 Computing the terminal distance matrix

Firstly, the `termverttab` array, meant to summarize information on all terminal domains, is created. It holds, in an arbitrary order depending both on the order of the vertices in the input vertex list and on the result of successive bipartitions: the label of the vertex associated with the terminal domain, the weight of the domain, and its terminal domain number.

Then, the distance matrix for terminals is computed. For each terminal domain taken as a root vertex, a breadth-first traversal of the input graph is performed.

Vertices yet to be processed are enqueued in queutab, and, once picked, their distance information is stored into disttax, which also acts as a flag array. Once all vertices have been processed, the lower part of the disttab array is copied into the termdisttab array.

Finally, the distance matrix and domain information must be extended from terminal domains to all domains, which is performed by calling the archDeco ArchBuild2() routine, which is also used by archDecoArchLoad().

7.2 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.

7.2.1 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 nsndidxtab. Each request comprises two numbers: the global fine number of the remote vertex for which the coarse number is seeked, 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 `ercvcnttab` 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.

7.3 `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 that 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.

7.3.1 `dgraphFold2()`

The `dgraphFold2()` routine creates the folded distributed graph from the provided initial distributed graph and a part value which indicates to which half of the processing elements the graph is to be folded. This routine may be called simultaneously by two threads which will each perform the folding to their respective part.

In the context of this routine, a processing element can be of any of the following three kinds: a sender, which belongs to the other part, and has to transfer all its vertex and edge data to receiver processes; a receiver, which belongs to the receiving part and has vertex data to receive from sender processes; and a receiver-sender, that is, a process belonging to the receiving part yet is overloaded so that it has to off-load some of its data to other receivers.

The first phase of the algorithm set the data structures. Firstly, it computes compact graph data from the input graph, if needed, since the routine expects the adjacency sub-arrays to be contiguous. Then, `dgraphFoldComm()` is called (see Section 7.3.2), to compute the communication pattern for graph data exchange. The output of this routine consists, among others, in `commvrttab`, an array of blocks of data that contain each: the number of vertices to be sent/received, and the sender/receiver process number. This array is traversed to accumulate the number of vertices to be received, in `fldvertlocnbr`. Once this size is known, folded vertex arrays can be allocated, as well as work data index arrays. These arrays, which depend on the number of exchanges per process, are:

`fldvertidxtab`
: Array of vertex buffer start indices for vertex data send/receive operations. For pure senders and pure receivers, the first index value in this array is always 0. For receiver-senders, it is the number of vertices kept on the processing element. It is created independently on sender and receiver processing elements, thanks to the data returned by `dgraphFoldComm()`.

`fldedgeidxtab`
: Array of edge buffer start indices for edge data send/receive operations. For pure senders and pure receivers, the first index value in this array is always be 0. For receiver-senders, it is the number of edge arcs kept on the processing element. This array is created locally by sender and receiver-sender processes; for pure receivers, it is created from the data received in their `fldedge cnttab` arrays from sender processes.

`fldedgecnttab`
: Array of edge buffer sizes for edge data send/receive operations. This array is created locally by sender and receiver-sender processes; for pure receivers, it is filled-in with data received from sender and receiver-sender processes.

`fldedgecnptab`
: This array is used by senders only. It is a copy of their `fldedgecnttab` array. Since the MPI standard prescribes that buffers used for asynchronous sending cannot be used during communication, the `fldedgecnptab` array is used for sending data, while the `fldedgecnttab` array can still be used to provide data to subsequent MPI asynchronous communication function calls.

`requtab`
: MPI asynchronous send or receive request array. Each type of data to be exchanged (number of edges to be exchanged, vertex indices, adjacency sub-arrays, etc.) will lead to the creation of `commnbr` requests. This array is of size `commmax` times the number of categories of requests, that is, `DGRAPH FOLDTAGNBR`. The first slot indices are used incrementally for communications that do not require to wait for further processing: all sends for senders and receiver-senders, and two kinds of data for pure receivers: `vnumloctab` and `dataloctab`, because these two are not needed to check the consistency of the produced graph.

In debug mode, a collective communication terminates this phase, to assess whether all computations and memory allocations went well.

In the second phase of the algorithm, asynchronous data exchange requests are launched and processed. At this stage, the `fldvertidxtab`, `fldedgeidxtab`, and `fldedgecnttab` arrays are built on sender and receiver-sender processing elements. By way of asynchronous communications, vertex sub-arrays are exchanged between sender and receiver processes. Also, the numbers of edges to be exchanged are transferred between the `fldedgecnptab` arrays of senders and the `fldedge cnttab` of pure receivers. Once this data is received and aggregated, pure receiver processing elements can allocate their `edgeloctab` edge array, and compute their `fldedgeidxtab` array, which will subsequently be used for edge data reception.

For senders, the `requtab` array is filled incrementally, using the `requnbr` index, with all their send requests. For pure receivers, only the beginning of the `requtab` array will be used incrementally, for data that does not require to wait before further processing: `vnumloctab` and `dataloctab`. For all other kinds of data, requests are placed in specific blocks of the `requtab` array, of size `commmax`. Because of the above, these request blocks start at block index 2.

In a third phase of the algorithm, intertwined with the second one, vertex and edge adjacency data is renumbered by receivers according to the global ordering of folded vertices, according to the adjustment data returned by `dgraphFold Comm()`. Each type of data to process is subject to a loop of `commnbr` iterations, which performs a `MPI_Waitany()` operation using the tag of the data type, and processes the received chunk of data, in any order.

Sender-receiver processing elements execute both the code of senders, to off-load their extra data to pure receivers, and the code of receivers, to participate in the construction of the folded graph. Before entering this second role, their `commnbr` variable is set to 0, since they do not expect any incoming communication.

Once all computations have taken place, followed by an optional `dgraph Check()`, a `MPI_Waitany()` is performed on all the requests stored incrementally in the `requtab` array, to complete all outstanding communications.

7.3.2 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, (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;

`commvrttab`
: 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 been exchanged (or which will remain in place on a receiver processing element);

`vertdlttab`
: 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 `vertdlttab` contain strongly related information, they are separate arrays, for the sake of memory locality. Indeed, `vertadj tab` 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 `vert dlttab` 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 `sort sndbas`, 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 `DGRAPHFOLDCOMMNBR`.

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`, `vertdlttab` and `slotsndtab` arrays, is compacted to remove empty slots and to form the final `vertadjtab` and `vertdlttab` 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 `vertdlttab`
: These two arrays have been presented above. They are created only for receiver processes, and will be filled concurrently. They are of size ((`commmax` + 1) $*$ `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 `vertdlttab[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 `vertdlttab[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 ((`commmax` + 1) $*$ `procfldnbr`) items. During the second stage of the algorithm, it is filled so that, for any non-empty communication slot `i` in `vertadjtab` and `vertdlttab`, 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 `dgraph FoldComm()` routine.

```
orgvertcnttab = { 20, 20, 20, 20, 20, 20, 20, 1908 }
partval = 1
vertglbmax = 1908
Proc [0] (SND) 20 -> 0 : { [4] <- 20 }
Proc [1] (SND) 20 -> 0 : { [5] <- 20 }
Proc [2] (SND) 20 -> 0 : { [6] <- 20 }
Proc [3] (SND) 20 -> 0 : { [6] <- 20 }
Proc [4] (RCV) 20 -> 512 : { [0] -> 20 }, { [7] -> 472 }
Proc [5] (RCV) 20 -> 512 : { [1] -> 20 }, { [7] -> 472 }
Proc [6] (RCV) 20 -> 512 : { [2] -> 20 }, { [7] -> 452 }, { [3] -> 20 }
Proc [7] (RSD) 1908 -> 512 : { [4] <- 472 }, { [5] <- 472 }, { [6] <- 452 }
commmax = 4
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.

```
orgvertcnttab = { 0, 0, 0, 20, 40, 40, 40, 100 }
partval = 1
vertglbmax = 100
Proc [0] (SND) 0 -> 0 :
Proc [1] (SND) 0 -> 0 :
Proc [2] (SND) 0 -> 0 :
Proc [3] (SND) 20 -> 0 : { [4] <- 20 }
Proc [4] (RCV) 40 -> 60 : { [3] -> 20 }
Proc [5] (RCV) 40 -> 60 : { [7] -> 20 }
Proc [6] (RCV) 40 -> 60 : { [7] -> 20 }
Proc [7] (RSD) 100 -> 60 : { [5] <- 20 }, { [6] <- 20 }
commmax = 4
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).

7.4 dmeshDgraphDual()

The `dmeshDgraphDual()` routine creates a dual distributed graph of type `Dgraph` from a distributed mesh of type `Dmesh`. It can be seen as the distributed-memory version of the `meshGraphDual()` routine. An edge will be created between two elements only if these elements have at least `noconbr` nodes in common.

At the time being, the `Dmesh` data structure only stores the adjacency from local element vertices to node vertices, using their global, based, numbering. Consequently, building the element-to-element connectivity operates in three phases: firstly, to redistribute element-to-node edge information so as to build the node-to-element adjacency of each node; secondly, to provide relevant node adjacencies to processes requiring them (possibly duplicating the same adjacency on multiple processes); this will allow, in a third phase, to build the element-to-element adjacency of each local element.

7.4.1 Determining the node vertex range

In a preliminary sweep over every local element-to-node edge array, the local maximum global node index `vnodlocmax` is computed. Then, by way of an all-reduce-max operation, the global maximum global node index `vnodglbmax` is obtained. If the node global indices are all used, then the global number of vertex nodes, `vnodglbnbr`, is equal to `vnodglbmax` − `baseval` + 1, as valid node vertex global indices range from `baseval` to `vnodglbmax`, included.

In debug mode, the local minimum global node index `vnodlocmin` is also computed, and all-reduced-min into `vnodglbmin`, which should be equal to `baseval`.

Knowing the global node vertex index range is necessary to evenly distribute node vertex data across all processes, assuming node vertices will have an equivalent number of neighbors overall. The absence of some node vertex indices in this range will not break the algorithm (isolated node vertices will be created in the first phase, which will not be propagated anywhere in the second phase), but may cause load imbalance when handling the node vertices on each process.

7.4.2 Creating node adjacencies

In order to build node vertex adjacencies across all processes, some all-to-all communication must take place, in order to send element-to-node edge data to the processes that will host the given node vertices, turning the gathered data into node-to-element data. All-to-all communication of edges will be controlled by four arrays of `int`'s, of size `procglbnbr` each: `esndcnttab`, the edge send count array; `esnddsptab`, the edge send displacement array; `ercvcnttab`, the edge receive count array; and `ercvdsptab`, the edge receive displacement array. The edge data to be sent will be placed into `esnddattab`, the edge send data array, while the received edge data will be available in `ercvdattab`, the edge receive data array.

In order to determine how many edges have to be sent to each process, per-process singly linked lists are built, by way of two arrays: `prfrloctab` ("(per-)process first (index), local array"), of size `procglbnbr` since there must be as many lists as there are destination processes, and `eeneloctax` ("element edge next (index), local based array"), of size $(2 * \texttt{eelmlocnbr})$ since each of the local element-to-node edges has to be chained to (only) one list, to be sent to the relevant process, and each chaining will require two data: the global element number (which could not be retrieved in $O(1)$ time else), and the edge index of the next edge in the chaining (which will be the sentinel value `-1` at the end of the list).

All the cells of `prfrloctab` are initialized with `-1`, the end-of-list sentinel, and all cells of `esndcnttab` are initialized to 0, as this array will be used to count the number of edges to send to each process.

Then, the adjacencies of all local element vertices are traversed. For each element-to-node edge of index e, the index p of the process which will holds the node vertex is computed in $O(1)$ time, using the `dmeshDgraphDualProcNum()` routine. The edge data is then chained at the head of the linked list for this process: `prfrloctab[`p`]` stores the index of the edge, while `eeneloctax[`$2 * e$`]` stores the element global index, and `eeneloctax[`$2 * e + 1$`]` receives the old value of `prfrloctab[`p`]`, to maintain the forward chaining. Also, `esndloctab[`p`]` is increased by 2, since two more data will be sent to p in the upcoming all-to-all exchange.

Then, the contents of `esndcnttab` are all-to-all exchanged to fill-in `ercvcnttab`, which indicates the amount of edge data to be received from each process; the sum of its cells gives `ercvdatsiz`, which amounts to twice the number of local node-to-element edges to be created. The `vnodloctax` and `enodloctax` arrays can then be allocated, to hold the node vertex indices and edge adjacency, respectively. Then, from `esndcnttab` and `ercvcnttab` are derived the displacement arrays `esnddsptab` and `ercvdsptab`, respectively. Then, the `esnddattab` and `ercvdattab` temporary arrays can be allocated, after those that will remain in memory longer.

Then, the per-process linked lists are traversed, and the element-to-node edge

data, now turned into node-to-element edge data, is copied into the esnddattab array, after which an all-to-allv data exchange makes it available in the ercvdat tab array of each process.

Then, the received edge array is traversed, to count in vnodloctax the number of edges per node vertex. Once this counting is done, the vnodloctax is turned into a displacement array, which will be used to place node-to-element edges at their proper place in enodloctax. After this, the the received edge array is traversed again to record the node-to-element edges in enodloctax, and the contents of vnodloctax are restored.

7.4.3 Making node adjacencies available to concerned elements

To create element-to-element adjacencies from element-to-node adjacencies, the node-to-element adjacencies of all nodes used as neighbors of some element vertex have to be copied to the process owning this element vertex. Hence, the same node adjacency may have to be sent to several processes at the same time. In order to determine to which process the adjacency of some node vertex has to be sent, one can take advantage of the order in which edge data have been received in the ercvdat tab array: the adjacency of a node has to be sent to some process p if the global node index of this node vertex appears in the sub-array of ercvdattab starting from index ercvdsptab[p] and ending before index ercvdsptab[$p+1$] (or ercv datsiz for the last sub-array). However, a node vertex adjacency needs only be sent once to any process, even if more than one of its local elements need it. To do so, a local node vertex flag array, vnflloctax, of size vnodlocnbr, will contain the most recent process number requesting the node vertex. Hence, a node vertex adjacency will only be copied once to the node adjacency send data array for this process. All cells of the flag array are initially set to -1, an invalid process number.

In a first pass across the ercvdattab array, the number of node data to be sent to each process is computed, and stored in the relevant cell of the nsndcnt tab ("node (data) send count") array. For each concerned node vertex, the number of data items to be sent is equal to two (the global number of the node, and its degree), plus the number of element neighbors of the node vertex. A node vertex v will be accounted for, for a given process p, only if vnflloctax[v] $< p$, and once the node vertex is accounted for, it is flagged by setting vnflloctax[v] to p.

Then, the contents of the nsndcnttab array are all-to-all exchanged, to produce the nrcvcnttab array. From these two can be derived the nsnddsptab and nrcvdsptab send and receive displacement arrays, respectively, and nsnd datsiz and nrcvdatsiz, the overall number of data to be sent and received, respectively. The two node data send and receive arrays, nsnddattab and nrcvdat tab, can be allocated with these prescribed sizes. The send array will be allocated last, since it will be freed first, as soon as the data exchange completes.

In a second pass across the ercvdattab array, the adjacencies of the nodes that are encountered for the first time in this pass are copied to the nsnddat tab array, one process after the other, using the start indices contained in the nsnd dsptab array. In order not to have to reset the flag array between the two passes, a node vertex v will be accounted for, for a given process p, only if vnflloctax[v] $<$ (procglbnbr$+p$), and once the node vertex is accounted for, it is flagged by setting vnflloctax[v] to (procglbnbr $+ p$).

Then, an all-to-allv data exchange makes the node adjacency data available in the nrcvdattab array of each process.

It is now necessary to make node adjacency available in $O(1)$ time. This is

made possible through a hash table hnodtab of type DmeshDgraphDualHash Node, which, for each concerned node vertex, will point to the start of this node data (that is, the node degree and node-to-element adjacency) in the nrcvdat tab array. Since this hash table will be static (that is, read-only and of immutable size) and must contain all the local nodes, its maximum load capacity is set to 50 % (and not 25 % as usually done in SCOTCH degree-related hash tables). Once this hash table array is allocated, the nrcvdattab is traversed to populate it.

7.4.4 Creating the element-to-element adjacencies

The last phase of the algorithm is the building of element-to-element adjacencies. This is performed through a second hash table, helmtab, of type DmeshDgraph DualHashEdge. Since the maximum degree of element-to-element adjacencies cannot be known in advance, this hash table may be resized dynamically, and will be loaded at 25 % capacity to minimize collisions. Its functioning, including resizing, is described in Section 6.8 of this manual.

The local distributed adjacency data for the dual graph will be placed into the vertloctax and edgeloctax arrays. Hence, prior to building the element-to-element adjacency, these arrays are allocated. Since the distributed graph will be compact, vertloctax is of size (velmlocnbr + 1). Since the number of edges cannot be estimated in advance, the size of the edgeloctax array, starting from a plausible size, may have to be dynamically increased during its filling-in, each time by 25 % more.

For each local element, the preexisting element-to-node adjacency is traversed and, for each of the neighbor nodes, the node-to-element adjacency is traversed in turn, being read from nrcvdattab from the index provided by hnodtab.

If the neighbor element is not yet present in helmtab for the current local element, it is added to the element hash table, with a neighbor count in the hash table equal to (noconbr−1), since one common node has already been found. If the neighbor element is already present in helmtab for the current local element, and its neighbor count in the hash table is strictly greater than zero, the neighbor count is decremented. If, in any of the two above cases, the neighbor count reaches zero, the neighbor element is added to the adjacency list of the current element in edge loctax; this latter array is enlarged whenever full. It will be downsized to its exact final size once all the edges have been created.

Once the vertloctax and edgeloctax arrays are complete, the dgraphBuild2 () routine is called, to finalize the construction of the distributed dual graph.

7.5 dorderCheck()

The dorderCheck() routine checks the consistency of a distributed ordering of type Dorder. Since distributed orderings are complex structures (see Section 6.2), checking their global consistency in a scalable way is not trivial.

After a distributed ordering is computed, every processing element contains a piece of the inverse permutation array, split into local fragments. Each fragment belongs to a DorderCblk node of type DORDERCBLKLEAF, in the form of a peri loctab array and an ordelocval index. The index is the start global node vertex index associated with the first value in the array, and array values represent inverse permutation values associated with consecutive indices starting from the provided one. This core inverse permutation data is supplemented with management data, such as the local and global number of node vertices in the ordering.

7.5.1 Checking management data

In order to perform checks on the inverse permutation, management data must be consistent, especially regarding the number of local and global vertex nodes in the permutation.

On each process, the list of column blocks is traversed, and the number of local node vertices of leaf column blocks is accumulated locally. It is compared with the preexisting local sum, and summed across all processes to recompute `vnodglb nbr`, the global number of node vertices in the ordering. A maximum reduction is also performed on these values to compute `vnodglbmax`, the size of the biggest chunk of data that might be sent from a process to another in the following phase, and to check the consistency of `baseval` across processes.

7.5.2 Checking permutations

To be scalable, the permutation checking algorithm does not perform any centralization of data. Therefore, the search for duplicate or missing indices has to be performed twice: a first time for the inverse permutation indices, and a second time for the inverse permutation values.

In a first phase, all permutation fragments present on a process are gathered into a single local array, `sortsndtab`, of size double of `vnodlocnbr`. This array contains pairs of integer values, made of the global index in the inverse permutation, and the index of the permuted node vertex. All these index values are verified to range from `baseval` to `vnodglbnbr` + `baseval` − 1, inclusive. Then, `sortsndtab` arrays are locally sorted by ascending first index, to prepare for the communication rounds that will follow. Indeed, while there are `vnodglbnbr` such pairs with valid indices spread across all processes, they may not collectively represent a valid permutation, e.g., in case of missing or duplicate indices.

In a second phase, to start checking the consistency of the permutation, a `flagloctab` array is created on each process, of a size equal to `vnodglbnbr`/`procglbnbr`, within one. Then, `procglbnbr` rounds of communication take place. During each round r, each process, of rank `proclocnum`, sends a single message to the process of rank (`proclocnum` + r)%`procglbnbr`. This message contains all the permutation pairs that the sender process holds and which are in the range managed by the receiver process in its `flagloctab` array. Message contents are in fact sub-arrays of the senders local `sortsndtab` arrays, the lower and upper bounds of which are computed by dichotomy, thanks to the arrays being sorted. Transmitting the permutation values along with the permutation indices is not necessary in itself for the algorithm, yet may help for subsequent debugging. Receiving processes read the contents of their message, and flag the received permutation indices in their `flagloctab` array. Duplicate indices are detected when trying to flag an already flagged index. Missing indices are detected after the last communication round is complete, by checking all `flagloc tab` arrays for un-flagged cells. Transmitting the permutation values along with the permutation indices is not necessary in itself for the algorithm, yet may help in subsequent debugging. The above algorithm guarantees that the inverse permutation fragments collected on every process cover all the node vertices in the distributed ordering.

To check the consistency of the inverse permutation data, the values of the pairs in the `sortsndtab` arrays are swapped, and the array is sorted again. Then, the whole flagging algorithm described above is performed anew. This guarantees that all inverse permutation indices are unique, and cover all the node vertices.

7.6 dorderTreeDist()

The dorderTreeDist() routine builds, on every process, the column block elimination tree of the given distributed ordering (see Section 6.2). It is the distributed counterpart of routine orderTree(). To perform its task, it must gather all column block data scattered on all processes, and index it globally. This reconciliation process is performed in four main phases.

7.6.1 Gathering locally-rooted column block data

Each processing element has a set of local column blocks, of type DorderCblk, which possess fragments of the global inverse permutation. Among these are locally-rooted column blocks, that is, column block tree nodes the proclocnum index of which is equal to the rank of the processing element.

Firstly, dorderCblkDist2() is called to compute the local number of locally-rooted column blocks, dblklocnbr, and the global number of such column blocks in the distributed ordering, dblkglbnbr. These two values are subsequently used to allocate all the data arrays that will be required by the different phases of the algorithm.

Then, the local numbers of locally-rooted column blocks, dblklocnbr, and the local number of column blocks (that is, inverse permutation fragments), ordeptr->cblklocnbr, are gathered on all processing elements, as duos of ints in the dblkcnttab ("data for column block") count array. In the corresponding MPI_Allgather() operation, the destination area, pointed to by dblkcnttab, is in fact of twice the declared size of this array: it is the combination of dblkcnttab and dblkdsptab. Gnums are converted to ints at this stage because these values will subsequently be used as count values in MPI functions, which are of type int.

The count data received in the combined dblkcnttab and dblkdsptab arrays is then processed in a sequence designed not to overlap with the boundaries of the declared arrays. The count values of local column blocks are accumulated in the cblkdsptab displacement array, while the count values in dblkcnttab are multiplied by 4, because the data to be gathered will consist in quartets of Gnum values. In a second loop, the dblkdsptab displacement array is generated from the newly updated dblkcnttab.

Then, for each locally-rooted column block, the dataloctab local send data array is filled in with quadruplets of data: the global index of the column block (using cblkdsptab as the base index for each processing element), the start index in the inverse permutation for the vertices contained in the column block, the global index of the father of the column block, and the number of vertices contained in the column block sub-tree. This data is then shared across all processing elements, in the dblkglbtab array.

7.6.2 Sorting column block data

This phase aims at preparing data index arrays for the next phase.

The first sorting array, srt1glbtab, takes the start index of the inverse permutation of the column block (its ordeglbval value) as key, and the global index of the column block as its value. Once this array is sorted in ascending order by way of a call to intSort2asc2(), the key is replaced by the value, and the value is replaced by the based global column block index. After the second sorting of this array, it provides, when traversed in order, for each global index of some column block, the position of this column block data in the dataglbtab array.

The second sorting array, `srt2glbtab`, takes the global index of the father column block as key, and the position of the column block data in the `dataglbtab` array. After sorting, it provides, when traversed in order, for each global index of a father column block, the position of a child column block in the `dataglbtab` array. Because of the sorting, the first cell of the array should now contain a "`-1`", which identifies the root column block. With the current implementation, this root should be unique.

7.6.3 Reconciling column block data

This phase aims at reconciling all the data gathered from the processing elements, using the sorted data index arrays created in the previous phase, to build a unified elimination tree.

The first loop propagates father global block indices to their children. In it, the `srt2glbtab` and `srt1glbtab` arrays are traversed jointly. The loop starts at index 1 for `srt2glbtab`, so that the "`-1`" father index is not taken into account, and at index 0 for `srt1glbtab`, which identifies children column blocks by their father index. When their keys match, it means that a column block has its father as the current father index; if so, the index is replaced in `dataglbtab` by the global column block number of the father.

The second loop updates the `srt2glbtab` array, with global index of column blocks as key, and their index in `dataglbtab` as value. After sorting, it provides, when traversed in order, the index to the data in `dataglbtab` for each column block index.

7.6.4 Filling-in of user arrays

In the last loop, the user-provided arrays are filled-in. This loop traverses jointly `srt1glbtab` and `srt2glbtab`, which have been sorted by ascending global column block index; hence, their key values must be absolutely identical. `srt1glbtab` provides the based index of the column block in the `treeglbtax` and `sizeglbtax` user arrays to fill, while `srt2glbtab` provides the index of block data in `dataglbtab`. The two user arrays are filled in with the global column block number of their father, and the size of the subtree, respectively.

8 Procedures for new developments and release

8.1 Adding methods to the libScotch library

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

8.1.1 What to add

There are currently seven 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`;
- 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]`;
- graph partitioning methods with separator overlap, in module `wgraph`.

Every method of these seven 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.

8.1.2 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 that 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,
- a 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.

8.1.3 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, preferrably 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 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 provided). 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).

8.1.4 Adding the new method to the Make compilation environment

In order to be compiled with the Make environment, the new method must files be added to the `Makefile` of the `src/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, preferrably 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 occurences 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.

8.1.5 Adding the new method to the CMake compilation environment

In order to be compiled with the MAKE environment, the new method files must be added to the `CMakeLists.txt` of the `src/libscotch` source directory.

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

8.2 Adding routines to the public API

The public API of SCOTCH exposes a set of routines which can be called by user programs. These public symbols are subject to specific procedures, in particular to rename them according to the users' needs.

- Implement the C interface of the routine in a `src/libscotch/library_*.c` source file. This source file must include `src/libscotch/module.h` , so that function renaming can take place whenever necessary;
- Whether appropriate, implement the Fortran interface of the routine within a `src/libscotch/library_*_f.c` source file. The routines in this file will call the former ones. This source file must include `src/libscotch/module.h` as well, so that function renaming can take place whenever necessary;
- Create a `SCOTCH_NAME_PUBLIC` macro in `src/libscotch/module.h` , so that function renaming can take place, for instance when symbols have to be suffixed;
- Create the relevant manual pages in the SCOTCH or PT-SCOTCH user's manual.

8.3 Release procedure

This section describes the procedure for releasing a new version of SCOTCH on the Inria GitLab repository: `https://gitlab.inria.fr/scotch/scotch` . All commands are relative to the "`./src`" directory of the SCOTCH distribution.

8.3.1 Removal of debugging flags

Verification of the absence of level-3 debugging flags (such definitions should contain the keyword "`BROL`" in the associated comment, as an alternate search keyword):

```
1 grep 'DEBUG_.\+3' *c | grep define
```

8.3.2 Symbol renaming

After compiling with "SCOTCH_RENAME", verification of the absence of unprotected names:

```
make scotch ptscotch esmumps ptesmumps
nm ../lib/libscotch.a | grep -v -e SCOTCH -e ESMUMPS -e " b " -e " d " -e "
     t " -e " U " -e "scotchf" -e "fprintf" -e "fscanf" -e "malloc" -e "
    realloc" -e "free" -e "memset" -e "random" -e "get_pc_thunk." -e " r .
    LC" | (sed -z 's/\n/#/g' ; echo -n '#') | sed 's/[a-zA-Z0-9_]\+[.]o
    [:]##//g' | sed 's/#/\n/g' > /tmp/brol.txt
nm ../lib/libptscotch.a | grep -v -e SCOTCH -e ESMUMPS -e " b " -e " d " -e
     " t " -e " U " -e "scotchf" -e "fprintf" -e "fscanf" -e "malloc" -e "
    realloc" -e "free" -e "memset" -e "random" -e "get_pc_thunk." -e " r .
    LC" | (sed -z 's/\n/#/g' ; echo -n '#') | sed 's/[a-zA-Z0-9_]\+[.]o
    [:]##//g' | sed 's/#/\n/g' >> /tmp/brol.txt
nm ../lib/libesmumps.a | grep -v -e SCOTCH -e ESMUMPS -e " b " -e " d " -e
    " t " -e " U " -e "scotchf" -e "fprintf" -e "fscanf" -e "malloc" -e "
    realloc" -e "free" -e "memset" -e "random" -e "get_pc_thunk." -e " r .
    LC" | (sed -z 's/\n/#/g' ; echo -n '#') | sed 's/[a-zA-Z0-9_]\+[.]o
    [:]##//g' | sed 's/#/\n/g' >> /tmp/brol.txt
nm ../lib/libptesmumps.a | grep -v -e SCOTCH -e ESMUMPS -e " b " -e " d " -
    e " t " -e " U " -e "scotchf" -e "fprintf" -e "fscanf" -e "malloc" -e "
    realloc" -e "free" -e "memset" -e "random" -e "get_pc_thunk." -e " r .
    LC" | (sed -z 's/\n/#/g' ; echo -n '#') | sed 's/[a-zA-Z0-9_]\+[.]o
    [:]##//g' | sed 's/#/\n/g' >> /tmp/brol.txt
more /tmp/brol.txt
```

8.3.3 Update of copyright year

Verification and possible modification of the copyright year, defined in the "SCOTCH_COPYRIGHT_STRING" macro:

```
emacs libscotch/module.h
```

In case the year is updated, create a commit with the message:

> Set year to 20XX in copyright string

8.3.4 Update of version number

For minor revisions:

```
emacs Makefile ../doc/src/version.tex ../CMakeLists.txt
```

Then, for major releases:

```
emacs ../doc/src/scotch/Makefile ../doc/src/ptscotch/Makefile ../doc/src/
    maint/Makefile ../INSTALL.txt ../README.txt
```

Create a commit with the message:

> Set revision marks to vX.Y.Z

8.3.5 Generation of documentation

Once the version number is changed in relevant files, generate the documentation:

```
cd ../doc/src/maint
make
make install
cd ../ptscotch
```

```
make
make install
cd ../scotch
make
make install
cd ../../../src
```

In case of change of major or minor version, remove the old documentation files and add the new ones:

```
cd ../doc/
git rm -f *X.Y*
git add *X'.Y'*
git commit *X.Y* *X'.Y'*
cd ../src
```

In case program interfaces have been modified, check that the man pages have been modified accordingly and integrate them in the documentation commit.

Create a commit with the message:

> Generate documentation for vX.Y.Z

8.3.6 Creation of the local tag

On the local master branch:

```
git tag vX.Y.Z
git push --tags origin
```

8.3.7 Merging

Check that the continuous integration went well.
In `fpellegr/scotch`, on the left, in tab "Merge request", clich on "New merge request", to `scotch/scotch`. Put version number as title (e.g.: "v7.0.7"), and, in the message, a summary and itemized list of the improvements brought by the release. Then, click to generate the merge request.
In `scotch/scotch`, on the left, in tab "Merge request", click on the new merge request. Click on "Approve", then on "Merge". A new (short) pipe-line is launched, this time in `scotch/scotch`.
If the pipe-line succeeds, click to finalize the merging.

8.3.8 Creation of the public tag

In the `scotch/scotch` home page, click on the icon with a label ("Tag"), then on "New tag". Put the version number as the tag name (e.g., "v7.0.7").

8.3.9 Generation of the asset

In the `scotch/scotch` home page, click on the icon with a rocket ("Releases"), then on "New release". Select the newly created tag. Put as comment the text of the merge request.