libScotch 7.0 Maintainer's Guide

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

February 4, 2022

Abstract

This document describes some internals of the libScotch library.

Contents

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 Naming conventions

All of the files of the Scotch project have been written using strict coding conventions, to ease maintenance and further extension by external contributors. Therefore, contributors must follow these coding conventions so as to ease the work of their followers.

2.1 Variables

Variables 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.

Common radicals are:

- `vert`: vertex.

- `velo`: vertex load.

- `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).
- `edlo`: edge (arc) load.
- `graf`: graph.

Common suffices are:

- `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`).
- `num`: 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 "not based", and `baseval` $\leq$ `vertnum` $<$ `vertnnd` if the index is "based", that it, counted starting from `baseval`. For number basing and array indexing, see Section 2.3.
- `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 2.3.
- `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, `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.

2.2 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.

2.3 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 2.1), 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 xxxxtab, we have xxxxtax = xxxxtab−baseval. Consequently xxxxtax[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] = ...;
}
```

3 Structure of the libScotch library

As seen in Section 2.2, 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.

4 Files and data structures

User-manageable file formats are described in the SCOTCHuser'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.

4.1 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.

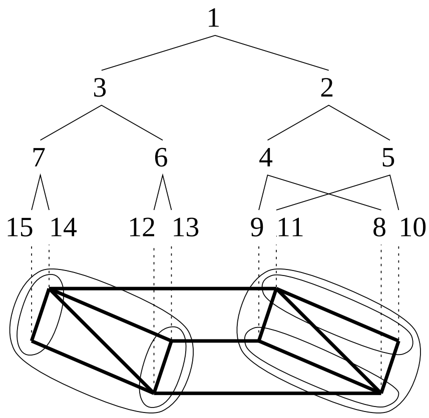
Also, while the "`deco 0`" format lists only the characteristics of terminal domains (in terms of weights and labels), the "`deco 1`" format provides these for all domains, so as to speed-up the retrieval of the size, weight and label of any domain, whether it is terminal or not.

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 $\mathrm{UB}(2,3)$, 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 numbers associated with every processor define a unique recursive bipartitioning of the target graph.

5 Adding a method to the libScotch library

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

5.1 What to add

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

- k-way graph mapping methods, in module `kgraph`,
- graph bipartitioning methods by means of edge separators, in module `bgraph`, used by the mapping method by dual recursive bipartitioning, implemented in `kgraph_map_rb.[ch]`,
- graph ordering methods, in module `hgraph`,
- graph separation methods by means of vertex separators, in module `vgraph`, used by the nested dissection ordering method implemented in `hgraph_order_nd.[ch]`,
- mesh ordering methods, in module `hmesh`,
- mesh separation methods with vertex separators, in module `vmesh`, used by the nested dissection ordering method implemented in `hmesh_order_nd.[ch]`.

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

```
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.

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.

5.2 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,
- an floating-point (`double`) type,
- an enumerated (`char`) type : this type is used to make a choice among a list of single character values, such as "`yn`". It is more readable than giving integer numerical values to method option flags,
- a strategy (Scotch `Strat` type) : a method can be passed a sub-strategy of a given type, which can be run on an augmented graph of the proper type. For

instance, the nested dissection method in hgraph_order_nd.c uses a graph separation strategy to compute its vertex separators.

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

5.4 Adding the new method to the makefile

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

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

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

6 Code explanations

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

6.1 `dgraphCoarsenBuild()`

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

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

6.1.1 Creating the fine-to-coarse vertex array

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

Hence, a loop on local multinode data fills `coardat.coargsttax`. The first local multinode vertex index is always local, by nature of the matching algorithm. If the second vertex is local too, `coardat.coargsttax` is filled instantly. Else, a request for the global coarse vertex number of the remote vertex is forged, in the `vsnddattab` array, indexed by the current index `coarsndidx` extracted from the neighbor process send index table `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.