PT-Scotch and libScotch 5.0
User's Guide

(version 5.0.4)

François Pellegrini
ScAlApplix project, INRIA Futurs
ENSEIRB & LaBRI, UMR CNRS 5800
Université Bordeaux I
351 cours de la Libération, 33405 TALENCE, FRANCE
pelegrin@labri.fr

December 8, 2007

Abstract

This document describes the capabilities and operations of PT-Scotch and libScotch, a software package and a software library which compute parallel sparse matrix block orderings of graphs. It gives brief descriptions of the algorithms, details the input/output formats, instructions for use, installation procedures, and provides a number of examples.

PT-Scotch is distributed as free/libre software, and has been designed such that new partitioning or ordering methods can be added in a straightforward manner. It can therefore be used as a testbed for the easy and quick coding and testing of such new methods, and may also be redistributed, as a library, along with third-party software that makes use of it, either in its original or in updated forms.

Contents

1 Introduction

1.1 Sparse matrix ordering

Many scientific and engineering problems can be modeled by sparse linear systems, which are solved either by iterative or direct methods. To achieve efficiency with direct methods, one must minimize the fill-in induced by factorization. This fill-in is a direct consequence of the order in which the unknowns of the linear system are numbered, and its effects are critical both in terms of memory and of computation costs.

Because there always exist large problem graphs which cannot fit in the memory of sequential computers and cost too much to partition, it is necessary to resort to parallel graph ordering tools. PT-Scotch provides such features.

1.2 Contents of this document

This document describes the capabilities and operations of PT-Scotch, a software package devoted to parallel sparse matrix block ordering. It is the parallel extension of Scotch, a sequential software package devoted to static mapping, graph and mesh partitioning, and sparse matrix block ordering. While both packages share a significant amount of code, because PT-Scotch transfers control to the sequential routines of the libScotch library when the subgraphs on which it operates are located on a single processor, the two sets of routines have a distinct user's manual. Readers interested in the sequential features of Scotch should refer to the Scotch *User's Guide* [25].

The rest of this manual is organized as follows. Section 2 presents the goals of the Scotch project, and section 3 outlines the most important aspects of the parallel partitioning and ordering algorithms that it implements. Section 4 defines the formats of the files used in PT-Scotch, section 5 describes the programs of the PT-Scotch distribution, and section 6 defines the interface and operations of the parallel routines of the libScotch library. Section 7 explains how to obtain and install the Scotch distribution. Finally, some practical examples are given in section 8.

2 The Scotch project

2.1 Description

Scotch is a project carried out at the *Laboratoire Bordelais de Recherche en Informatique* (LaBRI) of the Université Bordeaux I, and now within the ScALApplix project of INRIA Futurs. Its goal is to study the applications of graph theory to scientific computing, using a "divide and conquer" approach.

It focused first on static mapping, and has resulted in the development of the Dual Recursive Bipartitioning (or DRB) mapping algorithm and in the study of several graph bipartitioning heuristics [23], all of which have been implemented in the Scotch software package [26]. Then, it focused on the computation of high-quality vertex separators for the ordering of sparse matrices by nested dissection, by extending the work that has been done on graph partitioning in the context of static mapping [27, 28]. More recently, the ordering capabilities of Scotch have been extended to native mesh structures, thanks to hypergraph partitioning algorithms. New graph partitioning methods have also been recently added [6, 24]. Version 5.0 of Scotch is the first one to comprise parallel graph ordering routines.

2.2 Availability

Starting from version 4.0, which has been developed at INRIA within the ScAlApplix project, Scotch is available under a dual licensing basis. On the one hand, it is downloadable from the Scotch web page as free/libre software, to all interested parties willing to use it as a library or to contribute to it as a testbed for new partitioning and ordering methods. On the other hand, it can also be distributed, under other types of licenses and conditions, to parties willing to embed it tightly into closed, proprietary software.

The free/libre software license under which Scotch 5.0 is distributed is the CeCILL-C license [4], which has basically the same features as the GNU LGPL ("*Lesser General Public License*") [19]: ability to link the code as a library to any free/libre or even proprietary software, ability to modify the code and to redistribute these modifications. Version 4.0 of Scotch was distributed under the LGPL itself. This version did not comprise any parallel features.

Please refer to section 7 to see how to obtain the free/libre distribution of Scotch.

3 Algorithms

3.1 Parallel sparse matrix ordering by hybrid incomplete nested dissection

When solving large sparse linear systems of the form $Ax = b$, it is common to precede the numerical factorization by a symmetric reordering. This reordering is chosen in such a way that pivoting down the diagonal in order on the resulting permuted matrix PAP^T produces much less fill-in and work than computing the factors of A by pivoting down the diagonal in the original order (the fill-in is the set of zero entries in A that become non-zero in the factored matrix).

3.1.1 Hybrid incomplete nested dissection

The minimum degree and nested dissection algorithms are the two most popular reordering schemes used to reduce fill-in and operation count when factoring and solving sparse matrices.

The minimum degree algorithm [30] is a local heuristic that performs its pivot selection by iteratively selecting from the graph a node of minimum degree. It is known to be a very fast and general purpose algorithm, and has received much attention over the last three decades (see for example [1, 10, 21]). However, the algorithm is intrinsically sequential, and very little can be theoretically proved about its efficiency.

The nested dissection algorithm [11] is a global, recursive heuristic algorithm which computes a vertex set S that separates the graph into two parts A and B, ordering S with the highest remaining indices. It then proceeds recursively on parts A and B until their sizes become smaller than some threshold value. This ordering guarantees that, at each step, no non zero term can appear in the factorization process between unknowns of A and unknowns of B.

Many theoretical results have been obtained on nested dissection ordering [5, 20], and its divide and conquer nature makes it easily parallelizable. The main issue of the nested dissection ordering algorithm is thus to find small vertex separators that balance the remaining subgraphs as evenly as possible. Provided that good vertex separators are found, the nested dissection algorithm produces orderings which, both in terms of fill-in and operation count, compare favorably [12, 16, 27] to the ones obtained with the minimum degree algorithm [21]. Moreover, the elimination trees induced by nested dissection are broader, shorter, and better balanced, and therefore exhibit much more concurrency in the context of parallel Cholesky factorization [2, 8, 9, 12, 27, 29, and included references].

Due to their complementary nature, several schemes have been proposed to hybridize the two methods [15, 17, 27]. Our implementation is based on a tight coupling of the nested dissection and minimum degree algorithms, that allows each of them to take advantage of the information computed by the other [28].

However, because we do not provide a parallel implementation of the minimum degree algorithm, this hybridization scheme can only take place after enough steps of parallel nested dissection have been performed, such that the subgraphs to be ordered by minimum degree are centralized on individual processors.

3.1.2 Parallel ordering

The parallel computation of orderings in PT-Scotch involves three different levels of concurrency, corresponding to three key steps of the nested dissection process: the nested dissection algorithm itself, the multi-level coarsening algorithm used to compute separators at each step of the nested dissection process, and the refinement of the obtained separators. Each of these steps is described below.

Nested dissection As said above, the first level of concurrency relates to the parallelization of the nested dissection method itself, which is straightforward thanks to the intrinsically concurrent nature of the algorithm. Starting from the initial graph, arbitrarily distributed across p processors but preferably balanced in terms of vertices, the algorithm proceeds as illustrated in Figure 1 : once a separator has been computed in parallel, by means of a method described below, each of the p processors participates in the building of the distributed induced subgraph corresponding to the first separated part (even if some processors do not have any vertex of it). This induced subgraph is then folded onto the first $\lceil \frac{p}{2} \rceil$ processors, such that the average number of vertices per processor, which guarantees efficiency as it allows the shadowing of communications by a subsequent amount of computation, remains constant. During the folding process, vertices and adjacency lists owned by the $\lfloor \frac{p}{2} \rfloor$ sender processors are redistributed to the $\lceil \frac{p}{2} \rceil$ receiver processors so as to evenly balance their loads.

The same procedure is used to build, on the $\lfloor \frac{p}{2} \rfloor$ remaining processors, the folded induced subgraph corresponding to the second part. These two constructions being completely independent, the computations of the two induced subgraphs and their folding can be performed in parallel, thanks to the temporary creation of an extra thread per processor. When the vertices of the separated graph are evenly distributed across the processors, this feature favors load balancing in the subgraph building phase, because processors which do not have many vertices of one part will have the rest of their vertices in the other part, thus yielding the same overall workload to create both graphs in the same time. This feature can be disabled when the communication system of the target machine is not thread-safe.

At the end of the folding process, every processor has a folded subgraph fragment of one of the two folded subgraphs, and the nested dissection process car recursively proceed independently on each subgroup of $\frac{p}{2}$ (then $\frac{p}{4}$, $\frac{p}{8}$, etc.) processors, until each subgroup is reduced to a single processor. From then on, the nested dissection process will go on sequentially on every processor, using the nested dissection routines of the Scotch library, eventually ending in a coupling with minimum degree methods [28], as described in the previous section.

Graph coarsening The second level of concurrency concerns the computation of separators. The approach we have chosen is the now classical multi-level one [3, 14, 17]. It consists in repeatedly computing a set of increasingly coarser albeit topologically similar versions of the graph to separate, by finding matchings which collapse vertices and edges, until the coarsest graph obtained is no larger than a few hundreds of vertices, then computing a separator on this coarsest graph, and projecting back this separator, from coarser to finer graphs, up to the original graph. Most often, a local optimization algorithm, such as Kernighan-Lin [18] or Fiduccia-

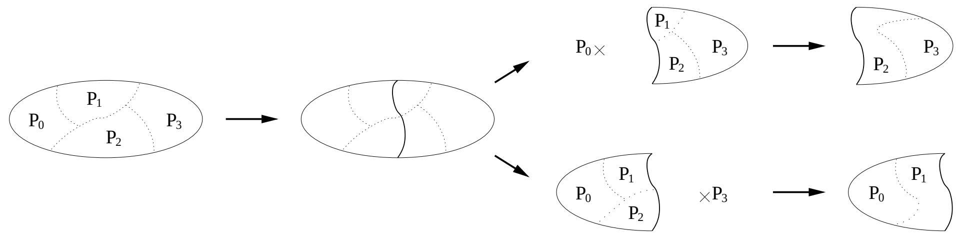

Figure 1: Diagram of a nested dissection step for a (sub-)graph distributed across four processors. Once the separator is known, the two induced subgraphs are built and folded (this can be done in parallel for both subgraphs), yielding two subgraphs, each of them distributed across two processors.

Mattheyses [7] (FM), is used in the uncoarsening phase to refine the partition that is projected back at every level, such that the granularity of the solution is the one of the original graph and not the one of the coarsest graph.

The main features of our implementation are outlined in Figure 2. Once the matching phase is complete, the coarsened subgraph building phase takes place. It can be parametrized so as to allow one to choose between two options. Either all coarsened vertices are kept on their local processors (that is, processors that hold at least one of the ends of the coarsened edges), as shown in the first steps of Figure 2, which decreases the number of vertices owned by every processor and speeds-up future computations, or else coarsened graphs are folded and duplicated, as shown in the next steps of Figure 2, which increases the number of working copies of the graph and can thus reduce communication and increase the final quality of the separators.

As a matter of fact, separator computation algorithms, which are local heuristics, heavily depend on the quality of the coarsened graphs, and we have observed with the sequential version of Scotch that taking every time the best partition among two ones, obtained from two fully independent multi-level runs, usually improved overall ordering quality. By enabling the folding-with-duplication routine (which will be referred to as "fold-dup" in the following) in the first coarsening levels, one can implement this approach in parallel, every subgroup of processors that hold a working copy of the graph being able to perform an almost-complete independent multi-level computation, save for the very first level which is shared by all subgroups, for the second one which is shared by half of the subgroups, and so on.

The problem with the fold-dup approach is that it consumes a lot of memory. Consequently, a good strategy can be to resort to folding only when the number of vertices of the graph to be considered reaches some minimum threshold. This threshold allows one to set a trade off between the level of completeness of the independent multi-level runs which result from the early stages of the fold-dup process, which impact partitioning quality, and the amount of memory to be used in the process.

Once all working copies of the coarsened graphs are folded on individual processors, the algorithm enters a multi-sequential phase, illustrated at the bottom of Figure 2: the routines of the sequential Scotch library are used on every processor to complete the coarsening process, compute an initial partition, and project it back up to the largest centralized coarsened graph stored on the processor. Then, the partitions are projected back in parallel to the finer distributed graphs, selecting the

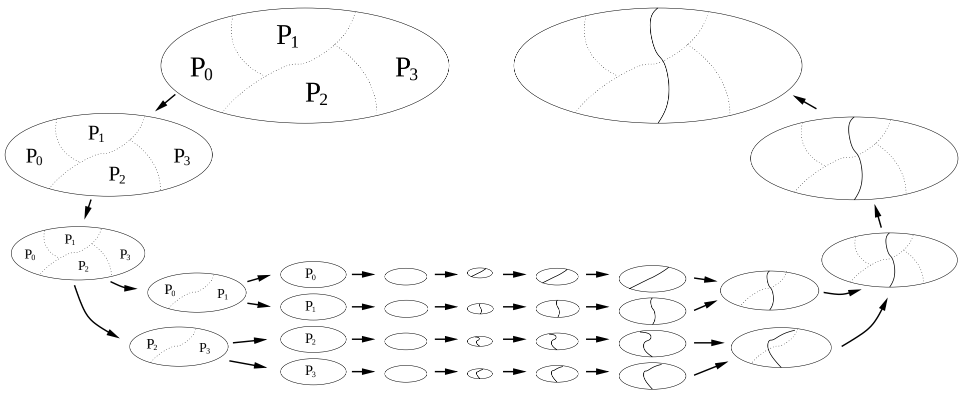

Figure 2: Diagram of the parallel computation of the separator of a graph distributed across four processors, by parallel coarsening with folding-with-duplication in the last stages, multi-sequential computation of initial partitions that are locally projected back and refined on every processor, and then parallel uncoarsening of the best partition encountered.

best partition between the two available when projecting to a level where fold-dup had been performed. This distributed projection process is repeated until we obtain a partition of the original graph.

Band refinement The third level of concurrency concerns the refinement heuristics which are used to improve the projected separators. At the coarsest levels of the multi-level algorithm, when computations are restricted to individual processors, the sequential FM algorithm of SCOTCH is used, but this class of algorithms does not parallelize well.

This problem can be solved in two ways: either by developing scalable and efficient local optimization algorithms, or by being able to use the existing sequential FM algorithm on very large graphs. In [6] has been proposed a solution which enables both approaches, and is based on the following reasoning. Since every refinement is performed by means of a local algorithm, which perturbs only in a limited way the position of the projected separator, local refinement algorithms need only to be passed a subgraph that contains the vertices that are very close to the projected separator.

The computation and use of distributed band graphs is outlined in Figure 3. Given a distributed graph and an initial separator, which can be spread across several processors, vertices that are closer to separator vertices than some small user-defined distance are selected by spreading distance information from all of the separator vertices, using our halo exchange routine. Then, the distributed band graph is created, by adding on every processor two anchor vertices, which are connected to the last layers of vertices of each of the parts. The vertex weight of the anchor vertices is equal to the sum of the vertex weights of all of the vertices they replace, to preserve the balance of the two band parts. Once the separator of the band graph has been refined using some local optimization algorithm, the new separator is projected back to the original distributed graph.

Basing on these band graphs, we have implemented a multi-sequential refinement algorithm, outlined in Figure 4. At every distributed uncoarsening step, a distributed band graph is created. Centralized copies of this band graph are then

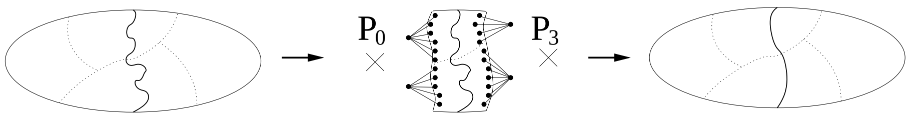

Figure 3: Creation of a distributed band graph. Only vertices closest to the separator are kept. Other vertices are replaced by anchor vertices of equivalent total weight, linked to band vertices of the last layer. There are two anchor vertices per processor, to reduce communication. Once the separator has been refined on the band graph using some local optimization algorithm, the new separator is projected back to the original distributed graph.

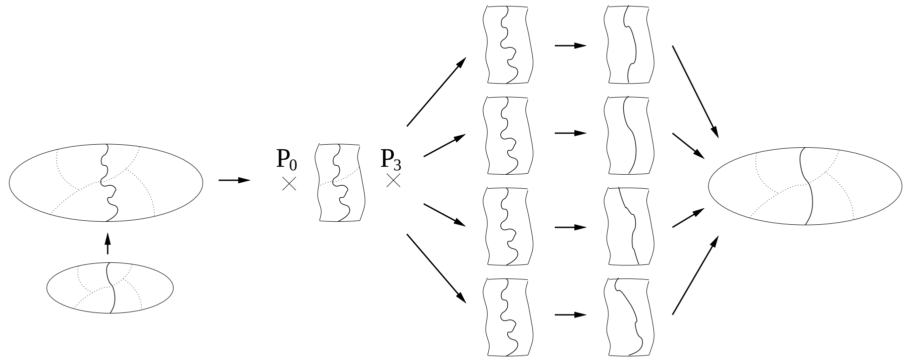

Figure 4: Diagram of the multi-sequential refinement of a separator projected back from a coarser graph distributed across four processors to its finer distributed graph. Once the distributed band graph is built from the finer graph, a centralized version of it is gathered on every participating processor. A sequential FM optimization can then be run independently on every copy, and the best improved separator is then distributed back to the finer graph.

gathered on every participating processor, which serve to run fully independent instances of our sequential FM algorithm. The perturbation of the initial state of the sequential FM algorithm on every processor allows us to explore slightly different solution spaces, and thus to improve refinement quality. Finally, the best refined band separator is projected back to the distributed graph, and the uncoarsening process goes on.

3.1.3 Performance criteria

The quality of orderings is evaluated with respect to several criteria. The first one, NNZ, is the number of non-zero terms in the factored reordered matrix. The second one, OPC, is the operation count, that is the number of arithmetic operations required to factor the matrix. The operation count that we have considered takes into consideration all operations (additions, subtractions, multiplications, divisions) required by Cholesky factorization, except square roots; it is equal to $\sum_c n_c^2$, where n_c is the number of non-zeros of column c of the factored matrix, diagonal included.

A third criterion for quality is the shape of the elimination tree; concurrency in parallel solving is all the higher as the elimination tree is broad and short. To measure its quality, several parameters can be defined: $h_{\min}$, $h_{\max}$, and h_{avg} denote the minimum, maximum, and average heights of the tree[1], respectively, and h_{dlt} is the variance, expressed as a percentage of h_{avg}. Since small separators result in small chains in the elimination tree, h_{avg} should also indirectly reflect the quality of separators.

4 Files and data structures

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 we 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 Distributed graph files

Because even very large graphs are most often stored in the form of centralized files, the distributed graph loading routine of the PT-SCOTCH package, as well as all parallel programs which handle distributed graphs, are able to read centralized graph files in the SCOTCH format and to scatter them on the fly across the available processors (the format of centralized SCOTCH graph files is described in the SCOTCH *User's Guide* [25]). However, in order to reduce loading time, a distributed graph format has been designed, so that the different file fragments which comprise distributed graph files can be read in parallel and be stored on local disks on the nodes of a parallel or grid cluster.

Distributed graph files, which usually end in "`.dgr`", describe fragments of valuated graphs, which can be valuated process graphs to be mapped onto target architectures, or graphs representing the adjacency structures of matrices to order.

In SCOTCH, graphs are represented by means of adjacency lists: the definition of each vertex is accompanied by the list of all of its neighbors, i.e. all of its adjacent arcs. Therefore, the overall number of edge data is twice the number of edges. Distributed graphs are stored as a set of files which contain each a subset of graph vertices and their adjacencies. The purpose of this format is to speed-up the loading and saving of large graphs when working for some time with the same number of processors: the distributed graph loading routine will allow each of the processors to read in parallel from a different file. Consequently, the number of files must be equal to the number of processors involved in the parallel loading phase.

The first line of a distributed graph file holds the distributed graph file version number, which is currently `2`. The second line holds the number of files across which the graph data is distributed (referred to as `procglbnbr` in LIBSCOTCH; see for instance Figure 6, page 21, for a detailed example), followed by the number of this file in the sequence (ranging from 0 to (`procglbnbr` − 1), and analogous to `procloc num` in Figure 6). The third line holds the global number of graph vertices (referred to as `vertglbnbr`), followed by the global number of arcs (inappropriately called `edgeglbnbr`, as it is in fact equal to twice the actual number of edges). The fourth line holds the number of vertices contained in this graph fragment (analogous to

[1]We do not consider as leaves the disconnected vertices that are present in some meshes, since they do not participate in the solving process.

vertlocnbr), followed by its local number of arcs (analogous to edgelocnbr). The fifth line holds three figures: the graph base index value (baseval), the starting global index for all vertices of this fragment (analogous to procdsptab[procloc num] in Figure 6) and a numeric flag.

The graph base index value records the value of the starting index used to describe the graph; it is usually 0 when the graph has been output by C programs, and 1 for Fortran programs. Its purpose is to ease the manipulation of graphs within each of these two environments, while providing compatibility between them.

The numeric flag, similar to the one used by the CHACO graph format [13], is made of three decimal digits. A non-zero value in the units indicates that vertex weights are provided. A non-zero value in the tenths indicates that edge weights are provided. A non-zero value in the hundredths indicates that vertex labels are provided; if it is the case, vertices can be stored in any order in the file; else, natural order is assumed, starting from the starting global index of each fragment.

This header data is then followed by as many lines as there are vertices in the graph fragment, that is, vertlocnbr lines. Each of these lines begins with the vertex label, if necessary, the vertex load, if necessary, and the vertex degree, followed by the description of the arcs. An arc is defined by the load of the edge, if necessary, and by the label of its other end vertex. The arcs of a given vertex can be provided in any order in its neighbor list. If vertex labels are provided, vertices can also be stored in any order in the file.

Figure 5 shows the contents of two complementary distributed graph files modeling a cube with unity vertex and edge weights and base 0, distributed across two processors.

```
2                               2
2       0                       2       1
8       24                      8       24
4       12                      4       12
0       000                     0       000
3       4       2       1       3       0       6       5
3       5       3       0       3       1       7       4
3       6       0       3       3       2       4       7
3       7       1       2       3       3       5       6
```

Figure 5: Two complementary distributed graph files representing a cube distributed across two processors.

5 Programs

5.1 Invocation

All of the programs comprised in the SCOTCH and PT-SCOTCH distributions have been designed to run in command-line mode without any interactive prompting, so that they can be called easily from other programs by means of "system ()" or "popen ()" system calls, or be piped together on a single shell command line. In order to facilitate this, whenever a stream name is asked for (either on input or output), the user may put a single "-" to indicate standard input or output. Moreover, programs read their input in the same order as stream names are given in the command line. It allows them to read all their data from a single stream (usually the standard input), provided that these data are ordered properly.

A brief on-line help is provided with all the programs. To get this help, use the "-h" option after the program name. The case of option letters is not significant, except when both the lower and upper cases of a letter have different meanings. When passing parameters to the programs, only the order of file names is significant; options can be put anywhere in the command line, in any order. Examples of use of the different programs of the PT-Scotch project are provided in section 8.

Error messages are standardized, but may not be fully explanatory. However, most of the errors you may run into should be related to file formats, and located in "...Load()" routines. In this case, compare your data formats with the definitions given in section 4, and use the `dgtst` program of the PT-Scotch distribution to check the consistency of distributed source graphs.

According to your MPI environment, you may either run the programs directly, or else have to invoke them by means of a command such as `mpirun`. Check your local MPI documentation to see how to specify the number of processors on which to run them.

5.2 File names

The programs of the PT-Scotch distribution can handle either the classical centralized Scotch graph files, or the distributed PT-Scotch graph files described in section 4.1.

In order to tell whether programs should read from, or write to, a single file located on only one processor, or to multiple instances of the same file on all of the processors, or else to distinct files on each of the processors, a special grammar has been designed, which is based on the "%" escape character. Four such escape sequences are defined, which are interpreted independently on every processor, prior to file opening. By default, when a filename is provided, it is assumed that the file is to be opened on only one of the processors, called the root processor, which is usually process 0 of the communicator within which the program is run. Using any of the first three escape sequences below will instruct programs to open in parallel a file of name equal to the interpreted filename, on every processor on which they are run.

`%p` Replaced by the number of processes in the global communicator in which the program is run. Leads to parallel opening.

`%r` Replaced on each process running the program by the rank of this process in the global communicator. Leads to parallel opening.

`%-` Discarded, but leads to parallel opening. This sequence is mainly used to instruct programs to open on every processor a file of identical name. The opened files can be, according whether the given path leads to a shared directory or to directories that are local to each processor, either to the opening of multiple instances of the same file, or to the opening of distinct files which may each have a different content, respectively (but in this latter case it is much recommended to identify files by means of the "%r" sequence).

`%%` Replaced by a single "%" character. File names using this escape sequence are not considered for parallel opening, unless one or several of the three other escape sequences are also present.

For instance, filename "`brol`" will lead to the opening of file "`brol`" on the root processor only, filename "`%-brol`" (or even "`br%-ol`") will lead to the parallel opening of files called "`brol`" on every processor, and filename "`brol%p-%r`" will lead to the opening of files "`brol2-0`" and "`brol2-1`", respectively, on each of the two processors on which which would run a program of the PT-Scotch distribution.

5.3 Description

5.3.1 `dgord`

Synopsis

`dgord` [*input_graph_file* [*output_ordering_file* [*output_log_file*]]] *options*

Description

The `dgord` program is the parallel sparse matrix block orderer. It uses an ordering strategy to compute block orderings of sparse matrices represented as source graphs, whose vertex weights indicate the number of DOFs per node (if this number is non homogeneous) and whose edges are unweighted, in order to minimize fill-in and operation count.

Since its main purpose is to provide orderings that exhibit high concurrency for parallel block factorization, it comprises a parallel nested dissection method [11], but sequential classical [21] and state-of-the-art [28] minimum degree algorithms are implemented as well, to be used on subgraphs located on single processors.

Ordering methods can be combined by means of selection, grouping, and condition operators, so as to define ordering strategies, which can be passed to the program by means of the `-o` option.

The *input_graph_file* filename can refer either to a centralized or to a distributed graph, according to the semantics defined in Section 5.2. The ordering file must be a centralized file.

Options

Since the program is devoted to experimental studies, it has many optional parameters, used to test various execution modes. Values set by default will give best results in most cases.

`-h` Display the program synopsis.

`-m`*output_mapping_file*

Write to *output_mapping_file* the mapping of graph vertices to column blocks. All of the separators and leaves produced by the nested dissection method are considered as distinct column blocks, which may be in turn split by the ordering methods that are applied to them. Distinct integer numbers are associated with each of the column blocks, such that the number of a block is always greater than the ones of its predecessors in the elimination process, that is, its descendants in the elimination tree. The structure of mapping files is described in detail in the relevant section of the Scotch *User's Guide* [25].

When the geometry of the graph is available, this mapping file may be processed by program `gout` to display the vertex separators and super-variable amalgamations that have been computed.

`-o`*strat*

Apply parallel ordering strategy *strat*. The format of parallel ordering strategies is defined in section 6.3.1.

`-r`*num*

Set the number of the root process which will be used for centralized file accesses. Set to 0 by default.

`-t`*output_tree_file*

Write to *output_tree_file* the structure of the separator tree. The data that is written resembles much the one of a mapping file: after a first line that contains the number of lines to follow, there are that many lines of mapping pairs, which associate an integer number with every graph vertex index. This integer number is the number of the column block which is the parent of the column block to which the vertex belongs, or -1 if the column block to which the vertex belongs is a root of the separator tree (there can be several roots, if the graph is disconnected).

Combined to the column block mapping data produced by option `-m`, the tree structure allows one to rebuild the separator tree.

`-V` Print the program version and copyright.

`-v`*verb*

Set verbose mode to *verb*, which may contain several of the following switches.

- `s` Strategy information. This parameter displays the default parallel ordering strategy used by `dgord`.
- `t` Timing information.

5.3.2 `dgscat`

Synopsis

`dgscat` [*input_graph_file* [*output_graph_file*]] *options*

Description

The `dgscat` program creates a distributed source graph, in the SCOTCH distributed graph format, from the given centralized source graph file.

The *input_graph_file* filename should therefore refer to a centralized graph, while *output_graph_file* must refer to a distributed graph, according to the semantics defined in Section 5.2.

Options

`-c` Check the consistency of the distributed graph at the end of the graph loading phase.

`-h` Display the program synopsis.

`-r`*num*

Set the number of the root process which will be used for centralized file accesses. Set to 0 by default.

`-V` Print the program version and copyright.

5.3.3 dgtst

Synopsis

`dgtst` [*input_graph_file* [*output_data_file*]] *options*

Description

The program `dgtst` is the source graph tester. It checks the consistency of the input source graph structure (matching of arcs, number of vertices and edges, etc.), and gives some statistics regarding edge weights, vertex weights, and vertex degrees.

It produces the same results as the `gtst` program of the Scotch sequential distribution.

Options

`-h` Display the program synopsis.

`-r`*num*
Set the number of the root process which will be used for centralized file accesses. Set to 0 by default.

`-V` Print the program version and copyright.

6 Library

All of the features provided by the programs of the PT-Scotch distribution may be directly accessed by calling the appropriate functions of the libScotch library, archived in files `ptlibscotch.a` and `libptscotcherr.a`. All of the existing parallel routines belong to three distinct classes:

- distributed source graph handling routines, which serve to declare, build, load, save, and check the consistency of distributed source graphs;
- strategy handling routines, which allow the user to declare and build parallel ordering strategies;
- parallel ordering routines, which allow the user to declare, compute, and save distributed orderings of distributed source graphs.

Error handling is performed using the existing sequential routines of the Scotch distribution, which are described in the Scotch *User's Guide* [25]. Their use is recalled in Section 6.8.

A ParMeTiS compatibility library, called `libptscotchparmetis.a`, is also available. It allows users who were previously using ParMeTiS in their software to take advantage of the efficieny of PT-Scotch without having to modify their code. The services provided by this library are described in Section 6.10.

6.1 Calling the routines of libScotch

6.1.1 Calling from C

All of the C routines of the libScotch library are prefixed with "`SCOTCH_`". The remainder of the function names is made of the name of the type of object to which the functions apply (e.g. "dgraph", "dorder", etc.), followed by the type of action performed on this object: "Init" for the initialization of the object, "Exit" for the freeing of its internal structures, "Load" for loading the object from one or several streams, and so on.

Typically, functions that return an error code return zero if the function succeeds, and a non-zero value in case of error.

For instance, the SCOTCH_dgraphInit and SCOTCH_dgraphLoad routines, described in section 6.4, can be called from C by using the following code.

```
#include "ptscotch.h"
  ...
  SCOTCH_Dgraph     grafdat;
  FILE *            fileptr;

  if (SCOTCH_dgraphInit (&grafdat) != 0) {
    ... /* Error handling */
  }
  if ((fileptr = fopen ("brol.grf", "r")) == NULL) {
  ... /* Error handling */
  }
  if (SCOTCH_dgraphLoad (&grafdat, fileptr, -1, 0) != 0) {
    ... /* Error handling */
  }
  ...
```

Although the "scotch.h" and "ptscotch.h" files may look very similar on your system, never mistake them, and always use the "ptscotch.h" file as the right include file for compiling a program which uses the parallel routines of the libScotch library, whether it also calls sequential routines or not.

6.1.2 Calling from Fortran

The routines of the libScotch library can also be called from Fortran. For any C function named SCOTCH_*typeAction*() which is documented in this manual, there exists a SCOTCHF*TYPEACTION*() Fortran counterpart, in which the separating underscore character is replaced by an "F". In most cases, the Fortran routines have exactly the same parameters as the C functions, save for an added trailing INTEGER argument to store the return value yielded by the function when the return type of the C function is not void.

Since all the data structures used in libScotch are opaque, equivalent declarations for these structures must be provided in Fortran. These structures must therefore be defined as arrays of DOUBLEPRECISIONs, of sizes given in file ptscotchf.h, which must be included whenever necessary.

For routines that read or write data using a FILE * stream in C, the Fortran counterpart uses an INTEGER parameter which is the numer of the Unix file descriptor corresponding to the logical unit from which to read or write. In most Unix implementations of Fortran, standard descriptors 0 for standard input (logical unit 5), 1 for standard output (logical unit 6) and 2 for standard error are opened by default. However, for files that are opened using OPEN statements, an additional function must be used to obtain the number of the Unix file descriptor from the number of the logical unit. This function is called FNUM in most Unix implementations of Fortran.

For instance, the SCOTCH_dgraphInit and SCOTCH_dgraphLoad routines, described in sections 6.4.1 and 6.4.4, respectively, can be called from Fortran by using the following code.

```
INCLUDE "ptscotchf.h"
DOUBLEPRECISION GRAFDAT(SCOTCH_DGRAPHDIM)
INTEGER RETVAL
...
CALL SCOTCHFDGRAPHINIT (GRAFDAT (1), RETVAL)
IF (RETVAL .NE. 0) THEN
...
OPEN (10, FILE='brol.grf')
CALL SCOTCHFDGRAPHLOAD (GRAFDAT (1), FNUM (10), 1, 0, RETVAL)
CLOSE (10)
IF (RETVAL .NE. 0) THEN
...
```

Although the "scotchf.h" and "ptscotchf.h" files may look very similar on your system, never mistake them, and always use the "ptscotchf.h" file as the include file for compiling a Fortran program that uses the parallel routines of the libScotch library, whether it also calls sequential routines or not.

All of the Fortran routines of the libScotch library are stubs which call their C counterpart. While this poses no problem for the usual integer and double precision data types, some conflicts may occur at compile or run time if your MPI implementation does not represent the MPI_Comm type in the same way in C and in Fortran. Please check on your platform to see in the mpi.h include file if the MPI_Comm data type is represented as an int. If it is the case, there should be no problem in using the Fortran routines of the PT-Scotch library.

6.1.3 Compiling and linking

The compilation of C or Fortran routines which use parallel routines of the libScotch library requires that either ptscotch.h or ptscotchf.h be included, respectively. Since some of the parallel routines of the libScotch library must be passed MPI communicators, it is necessary to include MPI files mpi.h or mpif.h, respectively, before the relevant PT-Scotch include files, such that prototypes of the parallel libScotch routines are properly defined.

The parallel routines of the libScotch library, along with taylored versions of the sequential routines, are grouped in a library file called libptscotch.a. Default error routines that print an error message and exit are provided in the classical Scotch library file libptscotcherr.a.

Therefore, the linking of applications that make use of the libScotch library with standard error handling is carried out by using the following options: "-lptscotch -lptscotcherr -lmpi -lm". The "-lmpi" option is most often not necessary, as the MPI library is automatically considered when compiling with commands such as mpicc.

If you want to handle errors by yourself, you should not link with library file libptscotcherr.a, but rather provide a SCOTCH_errorPrint() routine. Please refer to Section 6.8 for more information on error handling.

6.2 Data formats

All of the data used in the LIBSCOTCH interface are of integer type `SCOTCH_Num`. To hide the internals of PT-SCOTCH to callers, all of the data structures are opaque, that is, declared within `ptscotch.h` as dummy arrays of double precision values, for the sake of data alignment. Accessor routines, the names of which end in "`Size`" and "`Data`", allow callers to retrieve information from opaque structures.

In all of the following, whenever arrays are defined, passed, and accessed, it is assumed that the first element of these arrays is always labeled as `baseval`, whether `baseval` is set to 0 (for C-style arrays) or 1 (for Fortran-style arrays). PT-SCOTCH internally manages with base values and array pointers so as to process these arrays accordingly.

6.2.1 Distributed graph format

In PT-SCOTCH, distributed source graphs are represented so as to distribute graph data without any information duplication which could hinder scalability. The only data which are replicated on every process are of a size linear in the number of processes and small. Apart from these, the sum across all processes of all of the vertex data is in $O(v + p)$, where v is the overall number of vertices in the distributed graph and p the number of processes, and the sum of all of the edge data is in $O(e)$, where e is the overall number of arcs (that is, twice the number of edges) in the distributed graph. When graphs are ill-distributed, the overall halo vertex information may also be in $o(e)$ at worst, which makes the distributed graph structure fully scalable.

Distributed source graphs are described by means of adjacency lists. The description of a distributed graph requires several `SCOTCH_Num` scalars and arrays, as shown for instance in Figures 6 and 7. Some of these data are said to be global, and are duplicated on every process that holds part of the distributed graph; their names contain the "`glb`" infix. Others are local, that is, their value may differ for each process; their names contain the "`loc`" or "`gst`" infix. Global data have the following meaning:

`baseval`
: Base value for all array indexings.

`vertglbnbr`
: Overall number of vertices in the distributed graph.

`edgeglbnbr`
: Overall number of arcs in the distributed graph. Since edges are represented by both of their ends, the number of edge data in the graph is twice the number of edges.

`procglbnbr`
: Overall number of processes that share distributed graph data.

`proccnttab`
: Array holding the current number of local vertices borne by every process.

`procvrttab`
: Array holding the global indices from which the vertices of every process are numbered. For optimization purposes, this array has an extra slot which

stores a number which must be greater than all of the assigned global indices. For each process p, it must be ensured that $\texttt{procvrttab[}p+1\texttt{]} \geq (\texttt{procvrttab[}p\texttt{]} + \texttt{proccnttab[}p\texttt{]})$, that is, that no process can have more local vertices than allowed by its range of global indices. When the global numbering of vertices is continuous, for each process p, $\texttt{procvrttab[}p+1\texttt{]} = (\texttt{procvrttab[}p\texttt{]} + \texttt{proccnttab[}p\texttt{]})$.

Local data have the following meaning:

`vertlocnbr`

Number of local vertices borne by the given process. In fact, on every process p, `vertlocnbr` is equal to $\texttt{proccnttab[}p\texttt{]}$.

`vertgstnbr`

Number of both local and ghost vertices borne by the given process. Ghost vertices are local images of neighboring vertices located on distant processes.

`vertloctab`

Array of start indices in `edgeloctab` and `edgegsttab` of vertex adjacency sub-arrays.

`vendloctab`

Array of after-last indices in `edgeloctab` and `edgegsttab` of vertex adjacency sub-arrays. For any local vertex i, with $\texttt{baseval} \leq i < (\texttt{baseval} + \texttt{vertlocnbr})$, $\texttt{vendloctab[}i\texttt{]} - \texttt{vertloctab[}i\texttt{]}$ is the degree of vertex i.

When all vertex adjacency lists are stored in order in `edgeloctab` without any empty space between them, it is possible to save memory by not allocating the physical memory for `vendloctab`. In this case, illustrated in Figure 6, `vertloctab` is of size $\texttt{vertlocnbr} + 1$ and `vendloctab` points to $\texttt{vertloctab} + 1$. For these graphs , called "compact edge array graphs", or "compact graphs" for short, `vertloctab` is sorted in ascending order, $\texttt{vertloctab[baseval]} = \texttt{baseval}$ and $\texttt{vertloctab[baseval} + \texttt{vertlocnbr]} = (\texttt{baseval} + \texttt{edgelocnbr})$.

Since `vertloctab` and `vendloctab` only account for local vertices and not for ghost vertices, the sum across all processes of the sizes of these arrays does not depend on the number of ghost vertices; it is equal to $(v+p)$ for compact graphs and to $2v$ else.

`veloloctab`

Optional array, of size `vertlocnbr`, holding the integer load associated with every vertex.

`edgeloctab`

Array, of a size equal at least to $(\max_i(\texttt{vendloctab[}i\texttt{]}) - \texttt{baseval})$, holding the adjacency array of every local vertex. For any local vertex i, with $\texttt{baseval} \leq i < (\texttt{baseval} + \texttt{vertlocnbr})$, the global indices of the neighbors of i are stored in `edgeloctab` from $\texttt{edgeloctab[vertloctab[}i\texttt{]]}$ to $\texttt{edgeloctab[vendloctab[}i\texttt{]} - 1\texttt{]}$, inclusive.

Since ghost vertices do not have adjacency arrays, because only arcs from local vertices to ghost vertices are recorded and not the opposite, the overall sum of the sizes of all `edgeloctab` arrays is e.

`edgegsttab`
: Optional array holding the local and ghost indices of neighbors of local vertices. For any local vertex i, with `baseval` $\leq i <$ (`baseval` + `vertlocnbr`), the local and ghost indices of the neighbors of i are stored in `edgegsttab` from `edgegsttab[vertloctab[`i`]]` to `edgegsttab[vendloctab[`i`]`-1`]`, inclusive.

  Local vertices are numbered in global vertex order, starting from `baseval` to (`baseval` + `vertlocnbr` $-$ 1), inclusive. Ghost vertices are also numbered in global vertex order, from (`baseval`+`vertlocnbr`) to (`baseval`+`vertgstnbr`$-$ 1), inclusive.

  Only `edgeloctab` has to be provided by the user. `edgegsttab` is internally computed by PT-Scotch whenever needed, or can be explicitey asked for by the user by calling function `SCOTCH_dgraphGhst`. This array can serve to index user-defined arrays of quantities borne by graph vertices, which can be exchanged between neighboring processes thanks to the `SCOTCH_dgraphHalo` routine documented in Section 6.4.12.

`edloloctab`
: Optional array, of a size equal at least to ($\max_i$(`vendloctab[`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 `vendloctab` and `procvrttab` arrays. In order to dynamically manage distributed graphs, one just has to reserve index ranges large enough to create new vertices on each process, and to allocate `vertloctab`, `vendloctab` and `edgeloctab` arrays that are large enough to contain all of the expected new vertex and edge data. This can be done by passing `SCOTCH_graphBuild` a maximum number of local vertices, `vertlocmax`, greater than the current number of local vertices, `vertlocnbr`.

On every process p, vertices are globally labeled starting from `procvrttab[`p`]`, and locally labeled from `baseval`, leaving free space at the end of the local arrays. To remove some vertex of local index i, one just has to replace `vertloctab[`i`]` and `vendloctab[`i`]` with the values of `vertloctab[vertlocnbr` $-$ 1`]` and `vendloctab[vertlocnbr`$-$1`]`, respectively, and browse the adjacencies of all neighbors of former vertex (`vertlocnbr` $-$ 1) such that all (`vertlocnbr` $-$ 1) indices are turned into is. Then, `vertlocnbr` must be decremented, and `SCOTCH_dgraphBuild()` must be called to account for the change of topology. If a graph building routine such as `SCOTCH_dgraphLoad()` or `SCOTCH_dgraphBuild()` had already been called on the `SCOTCH_Dgraph` structure, `SCOTCH_dgraphFree()` has to be called first in order to free the internal structures associated with the older version of the graph, else these data would be lost, which would result in memory leakage.

To add a new vertex, one has to fill `vertloctab[vertnbr-1]` and `vendloctab[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 `vendloctab[`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 `vertloctab[`i`]` and `vendloctab[`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.

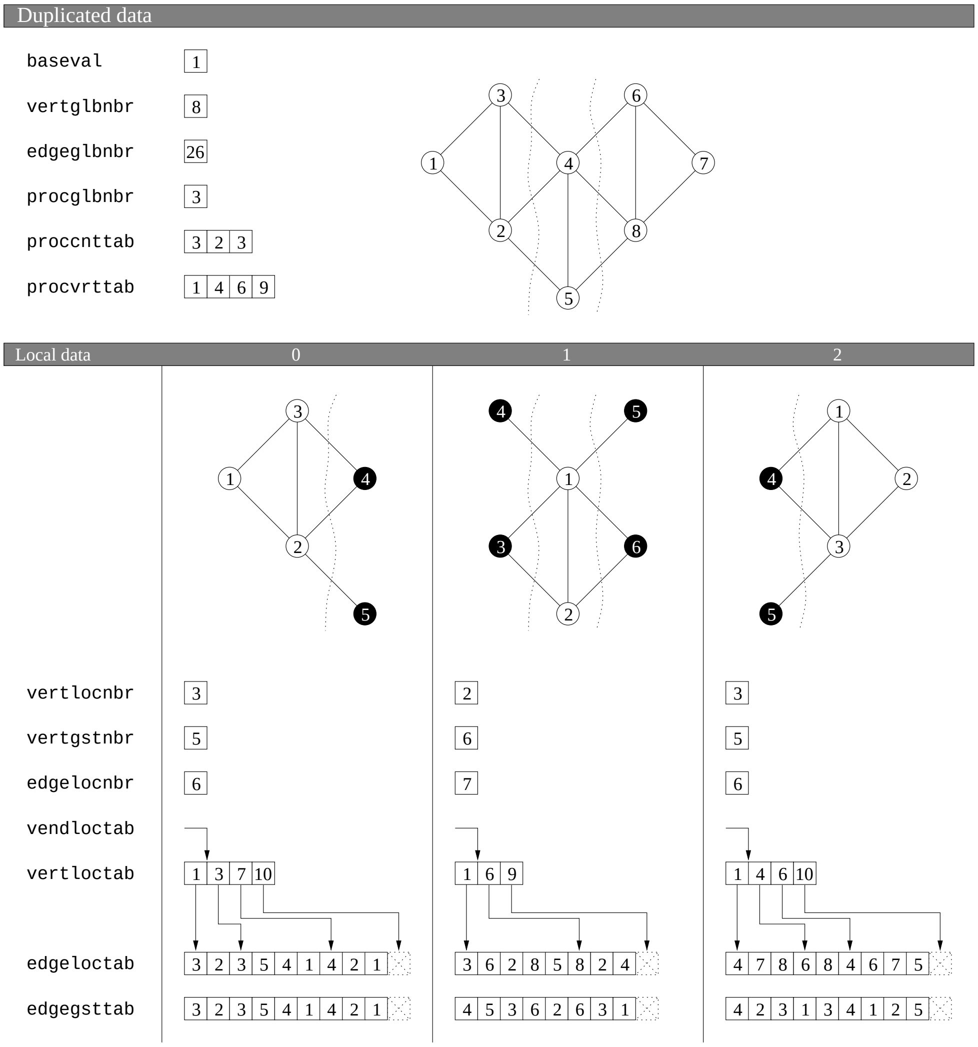

Figure 6: Sample distributed graph and its description by libScotch arrays using a continuous numbering and compact edge arrays. Numbers within vertices are vertex indices. Top graph is a global view of the distributed graph, labeled with global, continuous, indices. Bottom graphs are local views labeled with local and ghost indices, where ghost vertices are drawn in black. Since the edge array is compact, all `vertloctab` arrays are of size `vertlocnbr` + 1, and `vendloctab` points to `vertloctab` + 1. `edgeloctab` edge arrays hold global indices of end vertices, while optional `edgegsttab` edge arrays hold local and ghost indices. `veloloctab` and `edloloctab` are not represented.

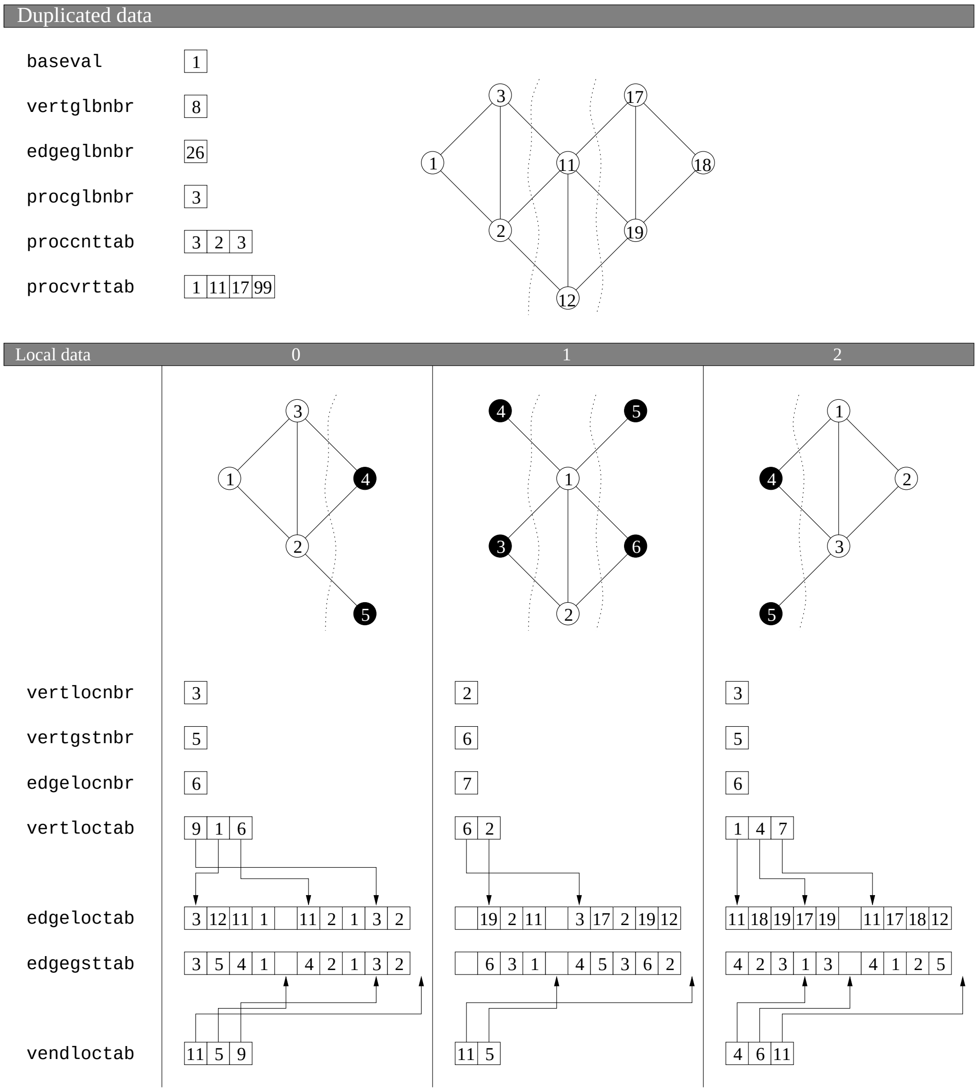

Figure 7: Adjacency structure of the sample graph of Figure 6 with a disjoint edge array and a discontinuous ordering. Both `vertloctab` and `vendloctab` are of size `vertlocnbr`. This allows for the handling of dynamic graphs, the structure of which can evolve with time.

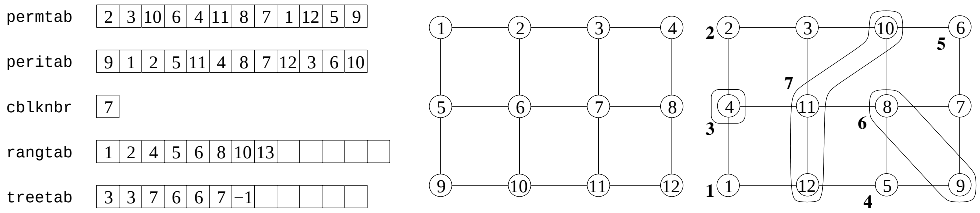

Figure 8: Arrays resulting from the ordering by complete nested dissection of a 4 by 3 grid based from 1. Leftmost grid is the original grid, and righmost grid is the reordered grid, with separators shown and column block indices written in bold.

6.2.2 Block ordering format

Block orderings associated with distributed graphs are described by means of block and permutation arrays, made of `SCOTCH_Num`s. In order for all orderings to have the same structure, irrespective of whether they are centralized or distributed, or whether they are created from graphs or meshes, all ordering data indices start from `baseval`. Consequently, row indices are related to vertex indices in memory in the following way: row i is associated with vertex i of the `SCOTCH_Dgraph` structure as if the vertex numbering used for the graph was continuous.

Block orderings are made of the following data:

`permtab`

: Array holding the permutation of the reordered matrix. Thus, if k = `permtab[`i`]`, then row i of the original matrix is now row k of the reordered matrix, that is, row i is the k^{th} pivot.

`peritab`

: Inverse permutation of the reordered matrix. Thus, if i = `peritab[`k`]`, then row k of the reordered matrix was row i of the original matrix.

`cblknbr`

: Number of column blocks (that is, supervariables) in the block ordering.

`rangtab`

: Array of ranges for the column blocks. Column block c, with `baseval` $\leq c <$ (`cblknbr` + `baseval`), contains columns with indices ranging from `rangtab[`i`]` to `rangtab[`$i+1$`]`, exclusive, in the reordered matrix. Therefore, `rangtab[baseval]` is always equal to `baseval`, and `rangtab[cblknbr + baseval]` is always equal to `vertglbnbr`+`baseval`. In order to avoid memory errors when column blocks are all single columns, the size of `rangtab` must always be one more than the number of columns, that is, `vertglbnbr` + 1.

`treetab`

: Array of ascendants of permuted column blocks in the separators tree. `treetab[i]` is the index of the father of column block i in the separators tree, or -1 if column block i is the root of the separators tree. Whenever separators or leaves of the separators tree are split into subblocks, as the block splitting, minimum fill or minimum degree methods do, all subblocks of the same level are linked to the column block of higher index belonging to the closest separator ancestor. Indices in `treetab` are based, in the same way as for the other blocking structures. See Figure 8 for a complete example.

6.3 Strategy strings

The behavior of the block ordering routines of the libScotch library is parametrized by means of strategy strings, which describe how and when given partitioning or ordering methods should be applied to graphs and subgraphs

6.3.1 Parallel ordering strategy strings

A parallel ordering strategy is made of one or several parallel ordering methods, which can be combined by means of strategy operators. The strategy operators that can be used in ordering strategies are listed below, by increasing precedence.

`(`*strat*`)`

Grouping operator. The strategy enclosed within the parentheses is treated as a single ordering method.

`/`*cond*`?`*strat1*`[:`*strat2*`];`

Condition operator. According to the result of the evaluation of condition *cond*, either *strat1* or *strat2* (if it is present) is applied. The condition applies to the characteristics of the current node of the separators tree, and can be built from logical and relational operators. Conditional operators are listed below, by increasing precedence.

cond1`|`*cond2*

Logical or operator. The result of the condition is true if *cond1* or *cond2* are true, or both.

cond1`&`*cond2*

Logical and operator. The result of the condition is true only if both *cond1* and *cond2* are true.

`!`*cond*

Logical not operator. The result of the condition is true only if *cond* is false.

var relop val

Relational operator, where *var* is a node variable, *val* is either a node variable or a constant of the type of variable *var*, and *relop* is one of '`<`', '`=`', and '`>`'. The node variables are listed below, along with their types.

`edge`

The global number of arcs of the current subgraph. Integer.

`levl`

The level of the subgraph in the separators tree, starting from zero for the initial graph at the root of the tree. Integer.

`load`

The overall sum of the vertex loads of the subgraph. It is equal to `vert` if the graph has no vertex loads. Integer.

`mdeg`

The maximum degree of the subgraph. Integer.

`proc`

The number of processes on which the current subgraph is distributed at this level of the separators tree. Integer.

`rank`

The rank of the current process among the group of processes on which the current subgraph is distributed at this level of the separators tree. Integer.

`vert`
: The global number of vertices of the current subgraph. Integer.

method[{*parameters*}]
: Parallel graph ordering method. Available parallel ordering methods are listed below.

The currently available parallel ordering methods are the following.

`n` Nested dissection method. The parameters of the nested dissection method are given below.

`ole=`*strat*
: Set the parallel ordering strategy that is used on every distributed leaf of the parallel separators tree if the node separation strategy `sep` has failed to separate it further.

`ose=`*strat*
: Set the parallel ordering strategy that is used on every distributed separator of the separators tree.

`osq=`*strat*
: Set the sequential ordering strategy that is used on every centralized subgraph of the separators tree, once the nested dissection process has gone far enough such that the number of processes handling some subgraph is restricted to one.

`sep=`*strat*
: Set the parallel node separation strategy that is used on every current leaf of the separators tree to make it grow. Parallel node separation strategies are described below, in section 6.3.2.

`s` Simple method. Vertices are ordered in their natural order. This method is fast, and should be used to order separators if the number of extra-diagonal blocks is not relevant

6.3.2 Parallel node separation strategy strings

A paralle node separation strategy is made of one or several parallel node separation methods, which can be combined by means of strategy operators. Strategy operators are listed below, by increasing precedence.

strat1 | *strat2*
: Selection operator. The result of the selection is the best vertex separator of the two that are obtained by the distinct application of *strat1* and *strat2* to the current separator.

strat1 strat2
: Combination operator. Strategy *strat2* is applied to the vertex separator resulting from the application of strategy *strat1* to the current separator. Typically, the first method used should compute an initial separation from scratch, and every following method should use the result of the previous one as a starting point.

`(`*strat*`)`

Grouping operator. The strategy enclosed within the parentheses is treated as a single separation method.

`/`*cond*`?`*strat1*[`:`*strat2*]`;`

Condition operator. According to the result of the evaluation of condition *cond*, either *strat1* or *strat2* (if it is present) is applied. The condition applies to the characteristics of the current subgraph, and can be built from logical and relational operators. Conditional operators are listed below, by increasing precedence.

cond1`|`*cond2*

Logical or operator. The result of the condition is true if *cond1* or *cond2* are true, or both.

cond1`&`*cond2*

Logical and operator. The result of the condition is true only if both *cond1* and *cond2* are true.

`!`*cond*

Logical not operator. The result of the condition is true only if *cond* is false.

var relop val

Relational operator, where *var* is a graph or node variable, *val* is either a graph or node variable or a constant of the type of variable *var*, and *relop* is one of '`<`', '`=`', and '`>`'. The graph and node variables are listed below, along with their types.

`edge`

The global number of edges of the current subgraph. Integer.

`levl`

The level of the subgraph in the separators tree, starting from zero for the initial graph at the root of the tree. Integer.

`load`

The overall sum of the vertex loads of the subgraph. It is equal to `vert` if the graph has no vertex loads. Integer.

`proc`

The number of processes on which the current subgraph is distributed at this level of the nested dissection process. Integer.

`rank`

The rank of the current process among the group of processes on which the current subgraph is distributed at this level of the nested dissection process. Integer.

`vert`

The number of vertices of the current subgraph. Integer.

The currently available parallel vertex separation methods are the following.

`b` Band method. Basing on the current distributed graph and on its partition, this method creates a new distributed graph reduced to the vertices which are at most at a given distance from the current separator, runs a parallel vertex separation strategy on this graph, and projects back the new separator to the current graph. This method is primarily used to run separator refinement methods during the uncoarsening phase of the multi-level parallel graph separation method. The parameters of the band method are listed below.

`strat=`*strat*
: Set the parallel vertex separation strategy to be applied to the band graph.

`width=`*val*
: Set the maximum distance from the current separator of vertices to be kept in the band graph. 0 means that only separator vertices themselves are kept, 1 that immediate neighboring vertices are kept too, and so on.

`m` Parallel vertex multi-level method. The parameters of the vertex multi-level method are listed below.

`asc=`*strat*
: Set the strategy that is used to refine the distributed vertex separators obtained at ascending levels of the uncoarsening phase by projection of the separators computed for coarser graphs. This strategy is not applied to the coarsest graph, for which only the `low` strategy is used.

`dlevl=`*nbr*
: Set the minimum level after which duplication is allowed in the folding process. A value of -1 results in duplication being always performed when folding.

`dvert=`*nbr*
: Set the average number of vertices per process under which the folding process is performed during the coarsening phase.

`low=`*strat*
: Set the strategy that is used to compute the vertex separator of the coarsest distributed graph, at the lowest level of the coarsening process.

`rat=`*rat*
: Set the threshold maximum coarsening ratio over which graphs are no longer coarsened. The ratio of any given coarsening cannot be less that 0.5 (case of a perfect matching), and cannot be greater than 1.0. Coarsening stops when either the coarsening ratio is above the maximum coarsening ratio, or the graph has fewer node vertices than the minimum number of vertices allowed.

`vert=`*nbr*
: Set the threshold minimum size under which graphs are no longer coarsened. Coarsening stops when either the coarsening ratio is above the maximum coarsening ratio, or the graph has fewer node vertices than the minimum number of vertices allowed.

`q` Multi-sequential method. The current distributed graph and its separator are centralized on every process that holds a part of it, and a sequential vertex separation method is applied independently to each of them. Then, the best separator found is projected back to the distributed graph. This method is primarily designed to operate on band graphs, which are orders of magnitude smaller than their parent graph. Else, memory bottlenecks are very likely to occur. The parameters of the multi-sequential method are listed below.

strat=*strat*

Set the sequential vertex separation strategy that is used to refine the separator of the centralized graph. For a description of all of the available sequential methods, please refer to the Scotch *User's Guide* [25].

z Zero method. This method moves all of the node vertices to the first part, resulting in an empty separator. Its main use is to stop the separation process whenever some condition is true.

6.4 Distributed graph handling routines

6.4.1 SCOTCH_dgraphInit

Synopsis

```
int SCOTCH_dgraphInit (SCOTCH_Dgraph *   grafptr,
                       MPI_Comm          comm)

scotchfdgraphinit (doubleprecision (*)   grafdat,
                   integer               comm,
                   integer               ierr)
```

Description

The SCOTCH_dgraphInit function initializes a SCOTCH_Dgraph structure so as to make it suitable for future parallel operations. It should be the first function to be called upon a SCOTCH_Dgraph structure. By accessing the communicator handle which is passed to it, SCOTCH_dgraphInit can know how many processes will be used to manage the distributed graph and can allocate its private structures accordingly.

SCOTCH_dgraphInit does not make a duplicate of the communicator which is passed to it, but instead keeps a reference to it, so that all future communications needed by libScotch to process this graph will be performed using this communicator. Therefore, it is the user's responsibility, whenever several libScotch routines might be called in parallel, to create appropriate duplicates of communicators so as to avoid any potential interferences between concurrent communications.

When the distributed graph is no longer of use, call function SCOTCH_dgraphExit to free its internal communication structures.

Return values

SCOTCH_dgraphInit returns 0 if the graph structure has been successfully initialized, and 1 else.

6.4.2 SCOTCH_dgraphExit

Synopsis

```
void SCOTCH_dgraphExit (SCOTCH_Dgraph *   grafptr)

scotchfdgraphexit (doubleprecision (*)   grafdat)
```

Description

The SCOTCH_dgraphExit function frees the contents of a SCOTCH_Dgraph structure previously initialized by SCOTCH_dgraphInit. All subsequent calls to SCOTCH_dgraph routines other than SCOTCH_dgraphInit, using this structure as parameter, may yield unpredictable results.

6.4.3 SCOTCH_dgraphFree

Synopsis

```
void SCOTCH_dgraphFree (SCOTCH_Dgraph *   grafptr)
scotchfdgraphfree (doubleprecision (*)   grafdat)
```

Description

The SCOTCH_dgraphFree function frees the graph data of a SCOTCH_Dgraph structure previously initialized by SCOTCH_dgraphInit, but preserves its internal communication data structures. This call is equivalent to a call to SCOTCH_dgraphExit immediately followed by a call to SCOTCH_dgraphInit with the same communicator as in the previous SCOTCH_dgraphInit call. Consequently, the given SCOTCH_Dgraph structure remains ready for subsequent calls to any distributed graph handling routine of the libScotch library.

6.4.4 SCOTCH_dgraphLoad

Synopsis

```
int SCOTCH_dgraphLoad (SCOTCH_Dgraph *   grafptr,
                       FILE *            stream,
                       SCOTCH_Num        baseval,
                       SCOTCH_Num        flagval)

scotchfdgraphload (doubleprecision (*)   grafdat,
                   integer               fildes,
                   integer               baseval,
                   integer               flagval,
                   integer               ierr)
```

Description

The SCOTCH_dgraphLoad routine fills the SCOTCH_Dgraph structure pointed to by grafptr with the centralized or distributed source graph description available from one or several streams stream in the Scotch graph formats (please refer to section 4.1 for a description of the distributed graph format, and to the Scotch *User's Guide* [25] for the centralized graph format).

When only one stream pointer is not null, the associated source graph file must be a centralized one, the contents of which are spread across all of the processes. When all stream pointers are non null, they can either refer to multiple instances of the same centralized graph, or to the distinct fragments of a distributed graph. In the first case, all processes read all of the contents of the centralized graph files but keep only the relevant part. In the second case, every process reads its fragment in parallel.

To ease the handling of source graph files by programs written in C as well as in Fortran, the base value of the graph to read can be set to `0` or `1`, by setting the `baseval` parameter to the proper value. A value of `-1` indicates that the graph base should be the same as the one provided in the graph description that is read from `stream`.

The `flagval` value is a combination of the following integer values, that may be added or bitwise-ored:

`0` Keep vertex and edge weights if they are present in the `stream` data.

`1` Remove vertex weights. The graph read will have all of its vertex weights set to one, regardless of what is specified in the `stream` data.

`2` Remove edge weights. The graph read will have all of its edge weights set to one, regardless of what is specified in the `stream` data.

Fortran users must use the `FNUM` function to obtain the number of the Unix file descriptor `fildes` associated with the logical unit of the graph file. Processes which would pass a `NULL` stream pointer in C must pass descriptor number `-1` in Fortran.

Return values

`SCOTCH_dgraphLoad` returns 0 if the distributed graph structure has been successfully allocated and filled with the data read, and 1 else.

6.4.5 `SCOTCH_dgraphSave`

Synopsis

```
int SCOTCH_dgraphSave (const SCOTCH_Dgraph *   grafptr,
                       FILE *                  stream)

scotchfdgraphsave (doubleprecision (*)   grafdat,
                   integer               fildes,
                   integer               ierr)
```

Description

The `SCOTCH_dgraphSave` routine saves the contents of the `SCOTCH_Dgraph` structure pointed to by `grafptr` to streams `stream`, in the SCOTCH distributed graph format (see section 4.1).

Fortran users must use the `FNUM` function to obtain the number of the Unix file descriptor `fildes` associated with the logical unit of the graph file.

Return values

`SCOTCH_dgraphSave` returns 0 if the graph structure has been successfully written to `stream`, and 1 else.

6.4.6 SCOTCH_dgraphBuild

Synopsis

```
int SCOTCH_dgraphBuild (SCOTCH_Dgraph *      grafptr,
                        const SCOTCH_Num     baseval,
                        const SCOTCH_Num     vertlocnbr,
                        const SCOTCH_Num     vertlocmax,
                        const SCOTCH_Num *   vertloctab,
                        const SCOTCH_Num *   vendloctab,
                        const SCOTCH_Num *   veloloctab,
                        const SCOTCH_Num *   vlblocltab,
                        const SCOTCH_Num     edgelocnbr,
                        const SCOTCH_Num     edgelocsiz,
                        const SCOTCH_Num *   edgeloctab,
                        const SCOTCH_Num *   edgegsttab,
                        const SCOTCH_Num *   edloloctab)
```

```
scotchfdgraphbuild (doubleprecision (*)   grafdat,
                    integer               baseval,
                    integer               vertlocnbr,
                    integer               vertlocmax,
                    integer (*)           vertloctab,
                    integer (*)           vendloctab,
                    integer (*)           veloloctab,
                    integer (*)           vlblloctab,
                    integer               edgelocnbr,
                    integer               edgelocsiz,
                    integer (*)           edgeloctab,
                    integer (*)           edgegsttab,
                    integer (*)           edloloctab,
                    integer               ierr)
```

Description

The `SCOTCH_dgraphBuild` routine fills the distributed source graph structure pointed to by `grafptr` with all of the data that are passed to it.

`baseval` is the graph base value for index arrays (typically 0 for structures built from C and 1 for structures built from Fortran). `vertlocnbr` is the number of local vertices on the calling process, used to create the `proccnttab` array. `vertlocmax` is the maximum number of local vertices to be created on the calling process, used to create the `procvrttab` array of global indices, and which must be set to `vertlocnbr` for graphs wihout holes in their global numbering. `vertloctab` is the local adjacency index array, of size (`vertlocnbr`+1) if the edge array is compact (that is, if `vendloctab` equals `vertloctab` + 1 or `NULL`), or of size `vertlocnbr` else. `vendloctab` is the adjacency end index array, of size `vertlocnbr` if it is disjoint from `vertloctab`. `veloloctab` is the local vertex load array, of size `vertlocnbr` if it exists. `vlblloctab` is the local vertex label array, of size `vertlocnbr` if it exists. `edgelocnbr` is the local number of arcs (that is, twice the number of edges), including arcs to local vertices as well as to ghost vertices. edgelocsiz is lower-bounded by the minimum size of the edge array required to encompass all used adjacency values; it is therefore at least equal to the maximum of the vendloctab entries, over all local vertices, minus baseval; it can be set to edgelocnbr if the edge array is compact. edgeloctab is the local adjacency array, of size at least edgelocsiz, which stores the global indices of end vertices. edgegsttab is the adjacency array, of size at least edgelocsiz, if it exists; if edgegsttab is given, it is assumed to be a pointer to an empty array to be filled with ghost vertex data computed by SCOTCH_dgraphGhst whenever needed by communication routines such as SCOTCH_dgraphHalo. edloloctab is the arc load array, of size edgelocsiz if it exists.

The vendloctab, veloloctab, vlblloctab, edloloctab and edgegsttab arrays are optional, and a null pointer can be passed as argument whenever they are not defined. Since, in Fortran, there is no null reference, passing the scotchfdgraphbuild routine a reference equal to vertloctab in the veloloctab or vlblloctab fields makes them be considered as missing arrays. The same holds for edloloctab and edgegsttab when they are passed a reference equal to edgeloctab. Setting vendloctab to refer to one cell after vertloctab yields the same result, as it is the exact semantics of a compact vertex array.

To limit memory consumption, SCOTCH_dgraphBuild does not copy array data, but instead references them in the SCOTCH_Dgraph structure. Therefore, great care should be taken not to modify the contents of the arrays passed to SCOTCH_dgraphBuild as long as the graph structure is in use. Every update of the arrays should be preceded by a call to SCOTCH_dgraphFree, to free internal graph structures, and eventually followed by a new call to SCOTCH_dgraphBuild to re-build these internal structures so as to be able to use the new distributed graph.

To ensure that inconsistencies in user data do not result in an erroneous behavior of the libScotch routines, it is recommended, at least in the development stage of your application code, to call the SCOTCH_dgraphCheck routine on the newly created SCOTCH_Dgraph structure before calling any other libScotch routine.

Return values

SCOTCH_dgraphBuild returns 0 if the graph structure has been successfully set with all of the input data, and 1 else.

6.4.7 SCOTCH_dgraphGather

Synopsis

```
int SCOTCH_dgraphGather (SCOTCH_Dgraph * const       dgrfptr,
                         const SCOTCH_Graph * const  cgrfptr)

scotchfdgraphgather (doubleprecision (*)  dgrfdat,
                     doubleprecision (*)  cgrfdat,
                     integer              ierr)
```

Description

The SCOTCH_dgraphGather routine gathers the contents of the distributed SCOTCH_Dgraph structure pointed to by dgrfptr to the centralized SCOTCH_Graph structure(s) pointed to by cgrfptr.

If only one of the processes has a non-null cgrfptr pointer, it is considered as the root process to which distributed graph data is sent. Else, all of the processes must provide a valid cgrfptr pointer, and each of them will receive a copy of the centralized graph.

Return values

SCOTCH_dgraphGather returns 0 if the graph structure has been successfully gathered, and 1 else.

6.4.8 SCOTCH_dgraphScatter

Synopsis

```
int SCOTCH_dgraphScatter (SCOTCH_Dgraph * const       dgrfptr,
                          const SCOTCH_Graph * const  cgrfptr)

scotchfdgraphscatter (doubleprecision (*)  dgrfdat,
                      doubleprecision (*)  cgrfdat,
                      integer              ierr)
```

Description

The SCOTCH_dgraphScatter routine scatters the contents of the centralized SCOTCH_Graph structure pointed to by cgrfptr across the processes of the distributed SCOTCH_Dgraph structure pointed to by dgrfptr.

Only one of the processes should provide a non-null cgrfptr parameter. This process is considered the root process for the scattering operation. Since, in Fortran, there is no null reference, processes which are not the root must indicate it by passing a pointer to the distributed graph structure equal to the pointer to their centralized graph structure.

The scattering is performed such that graph vertices are evenly spread across the processes of the communicator associated with the distributed graph, in ascending order. Every process receives either $\left\lceil \frac{\text{vertglbnbr}}{\text{procglbnbr}} \right\rceil$ or $\left\lfloor \frac{\text{vertglbnbr}}{\text{procglbnbr}} \right\rfloor$ vertices, according to its rank: processes of lower ranks are filled first, eventually with one more vertex than processes of higher ranks.

Return values

SCOTCH_dgraphScatter returns 0 if the graph structure has been successfully scattered, and 1 else.

6.4.9 SCOTCH_dgraphCheck

Synopsis

```
int SCOTCH_dgraphCheck (const SCOTCH_Dgraph *  grafptr)
```

```
scotchfdgraphcheck (doubleprecision (*)  grafdat,
                    integer              ierr)
```

Description

The SCOTCH_dgraphCheck routine checks the consistency of the given SCOTCH_Dgraph structure. It can be used in client applications to determine if a graph which has been created from user-generated data by means of the SCOTCH_dgraphBuild routine is consistent, prior to calling any other routines of the libScotch library which would otherwise return internal error messages or crash the program.

Return values

SCOTCH_dgraphCheck returns 0 if graph data are consistent, and 1 else.

6.4.10 SCOTCH_dgraphSize

Synopsis

```
void SCOTCH_dgraphSize (const SCOTCH_Dgraph *  grafptr,
                        SCOTCH_Num *           vertglbptr,
                        SCOTCH_Num *           vertlocptr,
                        SCOTCH_Num *           edgeglbptr,
                        SCOTCH_Num *           edgelocptr)

scotchfdgraphsize (doubleprecision (*)  grafdat,
                   integer              vertglbnbr,
                   integer              vertlocnbr,
                   integer              edgeglbnbr,
                   integer              edgelocnbr)
```

Description

The SCOTCH_dgraphSize routine fills the four areas of type SCOTCH_Num pointed to by vertglbptr, vertlocptr, edgeglbptr and edgelocptr with the number of global vertices and arcs (that is, twice the number of edges) of the given graph pointed to by grafptr, as well as with the number of local vertices and arcs borne by each of the calling processes.

Any of these pointers can be set to NULL on input if the corresponding information is not needed. Else, the reference to a dummy area can be provided, where all unwanted data will be written.

This routine is useful to get the size of a graph read by means of the SCOTCH_dgraphLoad routine, in order to allocate auxiliary arrays of proper sizes. If the whole structure of the graph is wanted, function SCOTCH_dgraphData should be preferred.

6.4.11 SCOTCH_dgraphData

Synopsis

```
void SCOTCH_dgraphData (const SCOTCH_Graph *  grafptr,
                        SCOTCH_Num *          baseptr,
                        SCOTCH_Num *          vertglbptr,
                        SCOTCH_Num *          vertlocptr,
                        SCOTCH_Num *          vertlocptz,
                        SCOTCH_Num *          vertgstptr,
                        SCOTCH_Num **         vertloctab,
                        SCOTCH_Num **         vendloctab,
                        SCOTCH_Num **         veloloctab,
                        SCOTCH_Num **         vlblloctab,
                        SCOTCH_Num *          edgeglbptr,
                        SCOTCH_Num *          edgelocptr,
                        SCOTCH_Num *          edgelocptz,
                        SCOTCH_Num **         edgeloctab,
                        SCOTCH_Num **         edgegsttab,
                        SCOTCH_Num **         edloloctab,
                        MPI_Comm *            comm)
```

```
scotchfdgraphdata (doubleprecision (*)  grafdat,
                   integer (*)          indxtab,
                   integer              baseval,
                   integer              vertglbnbr,
                   integer              vertlocnbr,
                   integer              vertlocmax,
                   integer              vertgstnbr,
                   integer              vertlocidx,
                   integer              vendlocidx,
                   integer              velolocidx,
                   integer              vlbllocidx,
                   integer              edgeglbnbr,
                   integer              edgelocnbr,
                   integer              edgelocsiz,
                   integer              edgelocidx,
                   integer              edgegstidx,
                   integer              edlolocidx,
                   integer              comm)
```

Description

The `SCOTCH_dgraphData` routine is the dual of the `SCOTCH_dgraphBuild` routine. It is a multiple accessor that returns scalar values and array references.

`baseptr` is the pointer to a location that will hold the graph base value for index arrays (typically 0 for structures built from C and 1 for structures built from Fortran). `vertglbptr` is the pointer to a location that will hold the global number of vertices. `vertlocptr` is the pointer to a location that will hold the number of local vertices. `vertlocptz` is the pointer to a location that will hold the maximum allowed number of local vertices, that is, (`procvrttab[`$p+1$`]` $-$ `procvrttab[`p`]`), where p is the rank of the local process. `vertgstptr` is the pointer to a location that will hold the number of local and ghost vertices if it has already been computed by a prior call to `SCOTCH_dgraphGhst`, and -1 else. `vertloctab` is the pointer to a location that will hold the reference to the adjacency index array, of size *vertlocptr + 1 if the adjacency array is compact, or of size *vertlocptr else. vendloctab is the pointer to a location that will hold the reference to the adjacency end index array, and is equal to vertloctab + 1 if the adjacency array is compact. veloloctab is the pointer to a location that will hold the reference to the vertex load array, of size *vertlocptr. vlblloctab is the pointer to a location that will hold the reference to the vertex label array, of size vertlocnbr. edgeglbptr is the pointer to a location that will hold the global number of arcs (that is, twice the number of global edges). edgelocptr is the pointer to a location that will hold the number of local arcs (that is, twice the number of local edges). edgelocptz is the pointer to a location that will hold the declared size of the local edge array, which must encompass all used adjacency values; it is at least equal to *edgelocptr. edgeloctab is the pointer to a location that will hold the reference to the local adjacency array of global indices, of size at least *edgelocptz. edgegsttab is the pointer to a location that will hold the reference to the ghost adjacency array, of size at least *edgelocptz; if it is non null, its data are valid if vertgstnbr is non-negative. edloloctab is the pointer to a location that will hold the reference to the arc load array, of size *edgelocptz. comm is the pointer to a location that will hold the MPI communicator of the distributed graph.

Any of these pointers can be set to NULL on input if the corresponding information is not needed. Else, the reference to a dummy area can be provided, where all unwanted data will be written.

Since there are no pointers in Fortran, a specific mechanism is used to allow users to access graph arrays. The scotchfdgraphdata routine is passed an integer array, the first element of which is used as a base address from which all other array indices are computed. Therefore, instead of returning references, the routine returns integers, which represent the starting index of each of the relevant arrays with respect to the base input array, or vertlocidx, the index of vertloctab, if they do not exist. For instance, if some base array myarray (1) is passed as parameter indxtab, then the first cell of array vertloc tab will be accessible as myarray(vertlocidx). In order for this feature to behave properly, the indxtab array must be word-aligned with the graph arrays. This is automatically enforced on most systems, but some care should be taken on systems that allow to access data that is not word-aligned. On such systems, declaring the array after a dummy doubleprecision array can coerce the compiler into enforcing the proper alignment. The integer value returned in comm is the communicator itself, not its index with respect to indxtab.

6.4.12 SCOTCH_dgraphHalo

Synopsis

```
int SCOTCH_dgraphHalo (SCOTCH_Dgraph * const  grafptr,
                       void *                 datatab,
                       MPI_Datatype           typeval)
```

```
scotchfdgraphhalo (doubleprecision (*)  grafdat,
                   doubleprecision (*)  datatab,
                   integer              typeval,
                   integer              ierr)
```

Description

The SCOTCH_dgraphHalo routine propagates the data borne by local vertices to all of the corresponding halo vertices located on neighboring processes. On every process, datatab should point to a data array of a size sufficient to hold vertgstnbr elements of the data type to be exchanged, the first vertlocnbr slots of which must already be filled with the information associated with the local vertices. On completion, the (vertgstnbr − vertlocnbr) remaining slots are filled with copies of the corresponding remote data obtained from the local parts of the data arrays of neighboring processes.

When the MPI data type to be used is not a collection of contiguous entries, great care should be taken in the definition of the upper bound of the type (by using the MPI_UB pseudo-datatype), such that when asking MPI to send a certain number of elements of the said type located at some address, contiguous areas in memory will be considered. Please refer to the MPI documentation regarding the creation of derived datatypes [22, Section 3.12.3] for more information.

To perform its data exchanges, the SCOTCH_dgraphHalo routine requires ghost vertex management data provided by the SCOTCH_dgraphGhst routine. Therefore, the edgegsttab array returned by the SCOTCH_dgraphData routine will always be valid after a call to SCOTCH_dgraphHalo.

Return values

SCOTCH_dgraphHalo returns 0 if halo data has been successfully exchanged, and 1 else.

6.4.13 SCOTCH_dgraphGhst

Synopsis

```
int SCOTCH_dgraphGhst (SCOTCH_Dgraph * const  grafptr)
scotchfdgraphghst (doubleprecision (*)  grafdat)
```

Description

The SCOTCH_dgraphGhst routine fills the edgegsttab arrays of the distributed graph structure pointed to by grafptr with the local and ghost vertex indices corresponding to the global vertex indices contained in its edgeloctab arrays, according to the semantics described in Section 6.2.1.

If memory areas had not been previously reserved by the user for the edgegsttab arrays and linked to the distributed graph structure through a call to SCOTCH_dgraphBuild, they are allocated. Their references can be retrieved on every process by means of a call to SCOTCH_dgraphData, which will also return the number of local and ghost vertices, suitable for allocating vertex data arrays for `SCOTCH_dgraphHalo`.

Return values

`SCOTCH_dgraphGhst` returns 0 if ghost vertex data has been successfully computed, and 1 else.

6.5 Distributed graph ordering routines

6.5.1 `SCOTCH_dgraphOrderInit`

Synopsis

```
int SCOTCH_dgraphOrderInit (const SCOTCH_Dgraph *  grafptr,
                            SCOTCH_Dordering *     ordeptr)

scotchfdgraphorderinit (doubleprecision (*)  grafdat,
                        doubleprecision (*)  ordedat,
                        integer              ierr)
```

Description

The `SCOTCH_dgraphOrderInit` routine initializes the distributed ordering structure pointed to by `ordeptr` so that it can be used to store the results of the parallel ordering of the associated distributed graph, to be computed by means of the `SCOTCH_dgraphOrderCompute` routine.

The `SCOTCH_dgraphOrderInit` routine should be the first function to be called upon a `SCOTCH_Dordering` structure for ordering distributed graphs. When the ordering structure is no longer of use, the `SCOTCH_dgraphOrderExit` function must be called, in order to to free its internal structures.

Return values

`SCOTCH_dgraphOrderInit` returns 0 if the distributed ordering structure has been successfully initialized, and 1 else.

6.5.2 `SCOTCH_dgraphOrderExit`

Synopsis

```
void SCOTCH_dgraphOrderExit (const SCOTCH_Dgraph *  grafptr,
                             SCOTCH_Dordering *     ordeptr)

scotchfgraphdorderexit (doubleprecision (*)  grafdat,
                        doubleprecision (*)  ordedat)
```

Description

The `SCOTCH_dgraphOrderExit` function frees the contents of a `SCOTCH_Dordering` structure previously initialized by `SCOTCH_dgraphOrderInit`. All subsequent calls to `SCOTCH_dgraphOrder*` routines other than `SCOTCH_dgraphOrderInit`, using this structure as parameter, may yield unpredictable results.

6.5.3 SCOTCH_dgraphOrderSave

Synopsis

```
int SCOTCH_dgraphOrderSave (const SCOTCH_Dgraph *     grafptr,
                            const SCOTCH_Dordering *  ordeptr,
                            FILE *                    stream)

scotchfdgraphordersave (doubleprecision (*)  grafdat,
                        doubleprecision (*)  ordedat,
                        integer              fildes,
                        integer              ierr)
```

Description

The SCOTCH_dgraphOrderSave routine saves the contents of the SCOTCH_Dordering structure pointed to by ordeptr to stream stream, in the SCOTCH ordering format. Please refer to the SCOTCH *User's Guide* [25] for more information about this format.

Since the ordering format is centralized, only one process should provide a valid output stream; other processes must pass a null pointer.

Fortran users must use the FNUM function to obtain the number of the Unix file descriptor fildes associated with the logical unit of the ordering file. Processes which would pass a NULL stream pointer in C must pass descriptor number -1 in Fortran.

Return values

SCOTCH_dgraphOrderSave returns 0 if the ordering structure has been successfully written to stream, and 1 else.

6.5.4 SCOTCH_dgraphOrderSaveMap

Synopsis

```
int SCOTCH_dgraphOrderSaveMap (const SCOTCH_Dgraph *     grafptr,
                               const SCOTCH_Dordering *  ordeptr,
                               FILE *                    stream)

scotchfgraphdordersavemap (doubleprecision (*)  grafdat,
                           doubleprecision (*)  ordedat,
                           integer              fildes,
                           integer              ierr)
```

Description

The SCOTCH_dgraphOrderSaveMap routine saves the block partitioning data associated with the SCOTCH_Dordering structure pointed to by ordeptr to stream stream, in the SCOTCH mapping format. A target domain number is associated with every block, such that all node vertices belonging to the same block are shown as belonging to the same target vertex. The resulting mapping file can be used by the `gout` program to produce pictures showing the different separators and blocks. Please refer to the SCOTCH *User's Guide* for more information on the SCOTCH mapping format and on `gout`.

Since the block partitioning format is centralized, only one process should provide a valid output stream; other processes must pass a null pointer.

Fortran users must use the `FNUM` function to obtain the number of the Unix file descriptor `fildes` associated with the logical unit of the ordering file. Processes which would pass a `NULL` stream pointer in C must pass descriptor number `-1` in Fortran.

Return values

`SCOTCH_dgraphOrderSaveMap` returns 0 if the ordering structure has been successfully written to `stream`, and 1 else.

6.5.5 SCOTCH_dgraphOrderSaveTree

Synopsis

```
int SCOTCH_dgraphOrderSaveTree (const SCOTCH_Dgraph *     grafptr,
                                const SCOTCH_Dordering *  ordeptr,
                                FILE *                    stream)

scotchfdgraphordersavetree (doubleprecision (*)  grafdat,
                            doubleprecision (*)  ordedat,
                            integer              fildes,
                            integer              ierr)
```

Description

The `SCOTCH_dgraphOrderSaveTree` routine saves the tree hierarchy information associated with the `SCOTCH_Dordering` structure pointed to by `ordeptr` to stream `stream`.

The format of the tree output file resembles the one of a mapping or ordering file: it is made up of as many lines as there are vertices in the ordering. Each of these lines holds two integer numbers. The first one is the index or the label of the vertex, and the second one is the index of its parent node in the separators tree, or -1 if the vertex belongs to a root node.

Since the tree hierarchy format is centralized, only one process should provide a valid output stream; other processes must pass a null pointer.

Fortran users must use the `FNUM` function to obtain the number of the Unix file descriptor `fildes` associated with the logical unit of the ordering file. Processes which would pass a `NULL` stream pointer in C must pass descriptor number `-1` in Fortran.

Return values

`SCOTCH_dgraphOrderSaveTree` returns 0 if the ordering structure has been successfully written to `stream`, and 1 else.

6.5.6 SCOTCH_dgraphOrderCompute

Synopsis

```
int SCOTCH_dgraphOrderCompute (const SCOTCH_Dgraph *  grafptr,
                               SCOTCH_Dordering *     ordeptr,
                               const SCOTCH_Strat *   straptr)
```

```
scotchfdgraphordercompute (doubleprecision (*)  grafdat,
                           doubleprecision (*)  ordedat,
                           doubleprecision (*)  stradat,
                           integer              ierr)
```

Description

The SCOTCH_dgraphOrderCompute routine computes in parallel a distributed block ordering of the distributed graph structure pointed to by grafptr, using the distributed ordering strategy pointed to by stratptr, and stores its result in the distributed ordering structure pointed to by ordeptr.

Return values

SCOTCH_dgraphOrderCompute returns 0 if the ordering has been successfully computed, and 1 else. In this latter case, the ordering arrays may however have been partially or completely filled, but their contents are not significant.

6.5.7 SCOTCH_dgraphOrderPerm

Synopsis

```
int SCOTCH_dgraphOrderPerm (const SCOTCH_Dgraph *  grafptr,
                            SCOTCH_Dordering *     ordeptr,
                            SCOTCH_Num *           permloctab)
```

```
scotchfdgraphorderperm (doubleprecision (*)  grafdat,
                        doubleprecision (*)  ordedat,
                        integer (*)          permloctab,
                        integer              ierr)
```

Description

The SCOTCH_dgraphOrderPerm routine fills the distributed direct permutation array permloctab according to the ordering provided by the given distributed ordering pointed to by ordeptr. Each permloctab local array should be of size vertlocnbr.

Return values

SCOTCH_dgraphOrderPerm returns 0 if the distributed permutation has been successfully computed, and 1 else.

6.5.8 SCOTCH_dgraphOrderCblkDist

Synopsis

```
SCOTCH_Num SCOTCH_dgraphOrderCblkDist (const SCOTCH_Dgraph *   grafptr,
                                       SCOTCH_Dordering *      ordeptr)

scotchfdgraphordercblkdist (doubleprecision (*)  grafdat,
                            doubleprecision (*)  ordedat,
                            integer              cblkglbnbr)
```

Description

The `SCOTCH_dgraphOrderCblkDist` routine returns on all processes the global number of distributed elimination tree (super-)nodes possessed by the given distributed ordering. Distributed elimination tree nodes are produced for instance by parallel nested dissection, before the ordering process goes sequential. Subsequent sequential nodes generated locally afterwards on individual processes are not accounted for in this figure.

This routine is used to allocate space for the tree structure arrays to be filled by the `SCOTCH_dgraphOrderTreeDist` routine.

Return values

`SCOTCH_dgraphOrderCblkDist` returns a positive number if the number of distributed elimination tree nodes has been successfully computed, and a negative value else.

6.5.9 SCOTCH_dgraphOrderTreeDist

Synopsis

```
int SCOTCH_dgraphOrderTreeDist (const SCOTCH_Dgraph *   grafptr,
                                SCOTCH_Dordering *      ordeptr,
                                SCOTCH_Num *            treeglbtab
                                SCOTCH_Num *            sizeglbtab)

scotchfdgraphordertreedist (doubleprecision (*)  grafdat,
                            doubleprecision (*)  ordedat,
                            integer (*)          treeglbtab,
                            integer (*)          sizeglbtab,
                            integer              ierr)
```

Description

The `SCOTCH_dgraphOrderTreeDist` routine fills on all processes the arrays representing the distributed part of the elimination tree structure associated with the given distributed ordering. This structure describes the sizes and relations between all distributed elimination tree (super-)nodes. These nodes are mainly the result of parallel nested dissection, before the ordering process goes sequential. Sequential nodes generated locally on individual processes are not represented in this structure.

A node can either be a leaf column block, which has no descendants, or a nested dissection node, which has most often three sons: its two separated sub-parts and the separator. A nested dissection node may have two sons only if the separator is empty; it cannot have only one son. Sons are indexed such that the separator of a block, if any, is always the son of highest index. Hence, the order of the indices of the two sub-parts matches the one of the direct permutation of the unknowns.

For any column block i, `treeglbtab[`i`]` holds the index of the father of node i in the elimination tree, or -1 if i is the root of the tree. All node indices start from `baseval`. `sizeglbtab[`i`]` holds the number of graph vertices possessed by node i, plus the ones of all of its descendants if it is not a leaf of the tree. Therefore, the `sizeglbtab` value of the root vertex is always equal to the number of vertices in the distributed graph.

Each of the `treeglbtab` and `sizeglbtab` arrays must be large enough to hold a number of `SCOTCH_Num`s equal to the number of distributed elimination tree nodes and column blocks, as returned by the `SCOTCH_dgraphOrderCblkDist` routine.

Return values

`SCOTCH_dgraphOrderTreeDist` returns 0 if the arrays describing the distributed part of the distributed tree structure have been successfully filled, and 1 else.

6.6 Centralized ordering handling routines

Since distributed ordering structures maintain scattered information which cannot be easily collated, the only practical way to access this information is to centralize it in a sequential `SCOTCH_Ordering` structure. Several routines are provided to create and destroy sequential orderings attached to a distributed graph, and to gather the information contained in a distributed ordering on such a sequential ordering structure.

Since the arrays which represent centralized ordering must be of a size equal to the global number of vertices, these routines are not scalable and may require much memory for very large graphs.

6.6.1 `SCOTCH_dgraphCorderInit`

Synopsis

```
int SCOTCH_dgraphCorderInit (const SCOTCH_Dgraph *  grafptr,
                             SCOTCH_Ordering *       cordptr,
                             SCOTCH_Num *            permtab,
                             SCOTCH_Num *            peritab,
                             SCOTCH_Num *            cblkptr,
                             SCOTCH_Num *            rangtab,
                             SCOTCH_Num *            treetab)
```

```
scotchfdgraphcorderinit (doubleprecision (*)  grafdat,
                         doubleprecision (*)  corddat,
                         integer (*)          permtab,
                         integer (*)          peritab,
                         integer              cblknbr,
                         integer (*)          rangtab,
                         integer (*)          treetab,
                         integer              ierr)
```

Description

The SCOTCH_dgraphCorderInit routine fills the centralized ordering structure pointed to by cordptr with all of the data that are passed to it. This routine is the equivalent of the SCOTCH_graphOrderInit routine of the Scotch sequential library, except that it takes a distributed graph as input. It is used to initialize a centralized ordering structure on which a distributed ordering will be centralized by means of the SCOTCH_dgraphOrderGather routine. Only the process on which distributed ordering data is to be centralized has to handle a centralized ordering structure.

permtab is the ordering permutation array, of size vertglbnbr, peritab is the inverse ordering permutation array, of size vertglbnbr, cblkptr is the pointer to a SCOTCH_Num that will receive the number of produced column blocks, rangtab is the array that holds the column block span information, of size vertglbnbr + 1, and treetab is the array holding the structure of the separators tree, of size vertglbnbr. Please refer to Section 6.2.2 for an explanation of their semantics. Any of these five output fields can be set to NULL if the corresponding information is not needed. Since, in Fortran, there is no null reference, passing a reference to grafptr will have the same effect.

The SCOTCH_dgraphCorderInit routine should be the first function to be called upon a SCOTCH_Ordering structure to be used for gathering distributed ordering data. When the centralized ordering structure is no longer of use, the SCOTCH_dgraphCorderExit function must be called, in order to to free its internal structures.

Return values

SCOTCH_dgraphCorderInit returns 0 if the ordering structure has been successfully initialized, and 1 else.

6.6.2 SCOTCH_dgraphCorderExit

Synopsis

```
void SCOTCH_dgraphCorderExit (const SCOTCH_Dgraph *  grafptr,
                              SCOTCH_Ordering *       cordptr)
```

```
scotchfdgraphcorderexit (doubleprecision (*)  grafdat,
                         doubleprecision (*)  corddat)
```

Description

The SCOTCH_dgraphCorderExit function frees the contents of a centralized SCOTCH_Ordering structure previously initialized by SCOTCH_dgraphCorderInit.

6.6.3 SCOTCH_dgraphOrderGather

Synopsis

```
int SCOTCH_dgraphOrderGather (const SCOTCH_Dgraph *  grafptr,
                              SCOTCH_Dordering *     cordptr,
                              SCOTCH_Ordering *      cordptr)

scotchfdgraphordergather (doubleprecision (*)  grafdat,
                          doubleprecision (*)  dorddat,
                          doubleprecision (*)  corddat,
                          integer              ierr)
```

Description

The SCOTCH_dgraphOrderGather routine gathers the distributed ordering data borne by dordptr to the centralized ordering structure pointed to by cordptr.

Return values

SCOTCH_dgraphOrderGather returns 0 if the centralized ordering structure has been successfully updated, and 1 else.

6.7 Strategy handling routines

This section presents basic strategy handling routines which are also described in the SCOTCH *User's Guide* but which are duplicated here for the sake of readability, as well as a strategy declaration routine which is specific to the PT-SCOTCH library.

6.7.1 SCOTCH_stratInit

Synopsis

```
int SCOTCH_stratInit (SCOTCH_Strat *  straptr)

scotchfstratinit (doubleprecision (*)  stradat,
                  integer              ierr)
```

Description

The SCOTCH_stratInit function initializes a SCOTCH_Strat structure so as to make it suitable for future operations. It should be the first function to be called upon a SCOTCH_Strat structure. When the strategy data is no longer of use, call function SCOTCH_stratExit to free its internal structures.

Return values

SCOTCH_stratInit returns 0 if the strategy structure has been successfully initialized, and 1 else.

6.7.2 SCOTCH_stratExit

Synopsis

```
void SCOTCH_stratExit (SCOTCH_Strat *  archptr)

scotchfstratexit (doubleprecision (*)  stradat)
```

Description

The SCOTCH_stratExit function frees the contents of a SCOTCH_Strat structure previously initialized by SCOTCH_stratInit. All subsequent calls to SCOTCH_strat routines other than SCOTCH_stratInit, using this structure as parameter, may yield unpredictable results.

6.7.3 SCOTCH_stratSave

Synopsis

```
int SCOTCH_stratSave (const SCOTCH_Strat *  straptr,
                      FILE *                stream)

scotchfstratsave (doubleprecision (*)  stradat,
                  integer              fildes,
                  integer              ierr)
```

Description

The SCOTCH_stratSave routine saves the contents of the SCOTCH_Strat structure pointed to by straptr to stream stream, in the form of a text string. The methods and parameters of the strategy string depend on the type of the strategy, that is, whether it is a bipartitioning, mapping, or ordering strategy, and to which structure it applies, that is, graphs or meshes.

Fortran users must use the FNUM function to obtain the number of the Unix file descriptor fildes associated with the logical unit of the output file.

Return values

SCOTCH_stratSave returns 0 if the strategy string has been successfully written to stream, and 1 else.

6.7.4 SCOTCH_stratDgraphOrder

Synopsis

```
int SCOTCH_stratDgraphOrder (SCOTCH_Strat *  straptr,
                             const char *    string)

scotchfstratdgraphorder (doubleprecision (*)  stradat,
                         character (*)        string,
                         integer              ierr)
```

Description

The `SCOTCH_stratDgraphOrder` routine fills the strategy structure pointed to by `straptr` with the distributed graph ordering strategy string pointed to by `string`. From this point, strategy `strat` can only be used as a distributed graph ordering strategy, to be used by function `SCOTCH_dgraphOrderCompute`. This routine must be called on every process with the same strategy string.

When using the C interface, the array of characters pointed to by `string` must be null-terminated.

Return values

`SCOTCH_stratDgraphOrder` returns 0 if the strategy string has been successfully set, and 1 else.

6.8 Error handling routines

The handling of errors that occur within library routines is often difficult, because library routines should be able to issue error messages that help the application programmer to find the error, while being compatible with the way the application handles its own errors.

To match these two requirements, all the error and warning messages produced by the routines of the libScotch library are issued using the user-definable variable-length argument routines `SCOTCH_errorPrint` and `SCOTCH_errorPrintW`. Thus, one can redirect these error messages to his own error handling routines, and can choose if he wants his program to terminate on error or to resume execution after the erroneous function has returned.

In order to free the user from the burden of writing a basic error handler from scratch, the `libptscotcherr.a` library provides error routines that print error messages on the standard error stream `stderr` and return control to the application. Application programmers who want to take advantage of them have to add `-lptscotcherr` to the list of arguments of the linker, after the `-lptscotch` argument.

6.8.1 SCOTCH_errorPrint

Synopsis

```
void SCOTCH_errorPrint (const char * const  errstr, ...  )
```

Description

The `SCOTCH_errorPrint` function is designed to output a variable-length argument error string to some stream.

6.8.2 SCOTCH_errorPrintW

Synopsis

```
void SCOTCH_errorPrintW (const char * const  errstr, ...)
```

Description

The SCOTCH_errorPrintW function is designed to output a variable-length argument warning string to some stream.

6.8.3 SCOTCH_errorProg

Synopsis

```
void SCOTCH_errorProg (const char *  progstr)
```

Description

The SCOTCH_errorProg function is designed to be called at the beginning of a program or of a portion of code to identify the place where subsequent errors take place. This routine is not reentrant, as it is only a minor help function. It is defined in libscotcherr.a and is used by the standalone programs of the SCOTCH distribution.

6.9 Miscellaneous routines

6.9.1 SCOTCH_randomReset

Synopsis

```
void SCOTCH_randomReset (void)
scotchfrandomreset ()
```

Description

The SCOTCH_randomReset routine resets the seed of the pseudo-random generator used by the graph partitioning routines of the LIBSCOTCH library. Two consecutive calls to the same LIBSCOTCH partitioning routines, and separated by a call to SCOTCH_randomReset, will always yield the same results, as if the equivalent standalone SCOTCH programs were used twice, independently, to compute the results.

6.10 ParMeTiS compatibility library

The ParMeTiS compatibility library provides stubs which redirect some calls to ParMeTiS routines to the corresponding PT-Scotch counterparts. In order to use this feature, the only thing to do is to re-link the existing software with the `lib ptscotchparmetis` library, and eventually with the original ParMeTiS library if the software uses ParMeTiS routines which do not need to have PT-Scotch equivalents, such as graph transformation routines. In that latter case, the "`-lptscotch parmetis`" argument must be placed before the "`-lparmetis`" one (and of course before the "`-lptscotch`" one too), so that routines that are redefined by PT-Scotch are chosen instead of their ParMeTiS counterpart. Routines of ParMeTiS which are not redefined by PT-Scotch may also require that the sequential MeTiS library be linked too. When no other ParMeTiS routines than the ones redefined by PT-Scotch are used, the "`-lparmetis`" argument can be omitted. See Section 8 for an example.

6.10.1 `ParMETIS_V3_NodeND`

Synopsis

```
void ParMETIS_V3_NodeND (const int * const  vtxdist,
                         const int * const  xadj,
                         const int * const  adjncy,
                         const int * const  numflag,
                         const int * const  options,
                         int * const        order,
                         int * const        sizes,
                         MPI_Comm *         comm)
```

```
parmetis_v3_nodend (integer (*)  vtxdist,
                    integer (*)  xadj,
                    integer (*)  adjncy,
                    integer      numflag,
                    integer (*)  options,
                    integer (*)  order,
                    integer (*)  sizes,
                    integer      comm)
```

Description

The `ParMETIS_V3_NodeND` function performs a nested dissection ordering of the distributed graph passed as arrays `vtxdist`, `xadj` and `adjncy`, using the default PT-Scotch ordering strategy. Unlike for ParMeTiS, this routine will compute an ordering even when the number of processors on which it is run is not a power of two. The `options` array is not used. When the number of processors is a power of two, the contents of the `sizes` array is equivalent to the one returned by the original `ParMETIS_V3_NodeND` routine, else it is filled with -1 values.

Users willing to get the tree structure of orderings computed on numbers of processors which are not power of two should use the native PT-Scotch ordering routine, and extract the relevant information from the distributed ordering with the SCOTCH_dgraphOrderCblkDist and SCOTCH_dgraphOrderTreeDist routines.

Similarly, as there is no ParMETIS_V3_NodeWND routine in ParMeTiS, users willing to order distributed graphs with node weights should directly call the PT-Scotch routines.

7 Installation

Version 5.0 of the Scotch software package, which contains the PT-Scotch routines, is distributed as free/libre software under the CeCILL-C free/libre software license [4], which is very similar to the LGPL. Therefore, it is no longer distributed as a set of binaries, but instead in the form of a source distribution, which can be downloaded from the Scotch web page at http://www.labri.fr/~pelegrin/scotch/ .

The extraction process will create a scotch_5.0 directory, containing several subdirectories and files. Please refer to the files called LICENSE_EN.txt or LICENCE_FR.txt, as well as file INSTALL_EN.txt, to see under which conditions your distribution of Scotch is licensed and how to install it.

To enable the use of POSIX threads in some routines, the SCOTCH_PTHREAD flag must be set. If your MPI implementation is not thread-safe, make sure this flag is not defined at compile time.

All Scotch users are welcome to send a mail to the author so that they can be added to the Scotch mailing list, and be automatically informed of new releases and publications.

8 Examples

This section contains chosen examples destined to show how the programs of the PT-Scotch project interoperate and can be combined. It is assumed that parallel programs are launched by means of the mpirun command, which comprises a -np option to set the number of processes on which to run them. Character "**%**" in bold represents the shell prompt.

- Create a distributed source graph file of 7 fragments from the centralized source graph file brol.grf stored in the current directory of process 0 of the MPI environment, and stores the resulting fragments in files labeled with the proper number of processors and processor ranks.

  ```
  % mpirun -np 7 dgscat brol.grf brol-%p-%r.dgr
  ```

- Compute on 3 processors the ordering of graph brol.grf, to be saved in a file called brol.ord written by process 0 of the MPI environment.

  ```
  % mpirun -np 7 dgord brol.grf brol.ord
  ```

- Compute on 4 processors the first three levels of nested dissection of graph brol.grf, and create an Open Inventor file called brol.iv to show the resulting separators and leaves.

```
% mpirun -np 4 dgord brol.grf /dev/null '-On{sep=/(levl<3)?m{
asc=b{strat=q{strat=f}},low=q{strat=h},seq=q{strat=m{low=h,asc=
b{strat=f}}}};,ole=s,ose=s,osq=n{sep=/(levl<3)?m{asc=b{strat=f},
low=h};}}' -mbrol.map
% gout brol.grf brol.xyz brol.map brol.iv
```

- Recompile a program that used ParMeTiS so that it uses PT-Scotch instead.

```
% mpicc brol.c -o brol -I${parmetisdir} -lptscotchparmetis
-lptscotch -lptscotcherr -lparmetis -lmetis -lm
```

Note that the "`-lptscotchparmetis`" option must be placed before the "`-lparmetis`" one, so that routines that are redefined by PT-Scotch are selected instead of their ParMeTiS counterpart. When no other ParMeTiS routines than the ones redefined by PT-Scotch are used, the "`-lparmetis -lmetis`" options can be omitted. The "`-I${parmetisdir}`" option may be necessary to provide the path to the original `parmetis.h` include file, which contains the prototypes of all of the ParMeTiS routines.

Credits

I wish to thank all of the following people :

- Cédric Chevalier, during his PhD at LaBRI, did research on efficient parallel matching algorithms and coded the parallel multi-level algorithm of PT-Scotch. He also studied parallel genetic refinement algorithms. Many thanks to him for the great job!

References

[1] P. Amestoy, T. Davis, and I. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. and Appl.*, 17:886–905, 1996.

[2] C. Ashcraft, S. Eisenstat, J. W.-H. Liu, and A. Sherman. A comparison of three column based distributed sparse factorization schemes. In *Proc. Fifth SIAM Conf. on Parallel Processing for Scientific Computing*, 1991.

[3] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6(2):101–117, 1994.

[4] CeCILL: "CEA-CNRS-INRIA Logiciel Libre" free/libre software license. Available from `http://www.cecill.info/licenses.en.html`.

[5] P. Charrier and J. Roman. Algorithmique et calculs de complexité pour un solveur de type dissections emboîtées. *Numerische Mathematik*, 55:463–476, 1989.

[6] C. Chevalier and F. Pellegrini. Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework. In *Proc. EuroPar, Dresden*, LNCS 4128, pages 243–252, September 2006.

[7] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181. IEEE, 1982.

[8] G. A. Geist and E. G.-Y. Ng. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18(4):291–314, 1989.

[9] A. George, M. T. Heath, J. W.-H. Liu, and E. G.-Y. Ng. Sparse Cholesky factorization on a local memory multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 9:327–340, 1988.

[10] A. George and J. W.-H. Liu. The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31:1–19, 1989.

[11] J. A. George and J. W.-H. Liu. *Computer solution of large sparse positive definite systems*. Prentice Hall, 1981.

[12] A. Gupta, G. Karypis, and V. Kumar. Scalable parallel algorithms for sparse linear systems. In *Proc. Stratagem'96, Sophia-Antipolis*, pages 97–110. INRIA, July 1996.

[13] B. Hendrickson and R. Leland. The Chaco user's guide. Technical Report SAND93–2339, Sandia National Laboratories, November 1993.

[14] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of Supercomputing*, 1995.

[15] B. Hendrickson and E. Rothberg. Improving the runtime and quality of nested dissection ordering. *SIAM J. Sci. Comput.*, 20(2):468–489, 1998.

[16] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report 95-035, University of Minnesota, June 1995.

[17] G. Karypis and V. Kumar. MeTiS – *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices – Version 4.0.* University of Minnesota, September 1998.

[18] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitionning graphs. *BELL System Technical Journal*, pages 291–307, February 1970.

[19] GNU Lesser General Public License. Available from `http://www.gnu.org/copyleft/lesser.html`.

[20] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal of Numerical Analysis*, 16(2):346–358, April 1979.

[21] J. W.-H. Liu. Modification of the minimum-degree algorithm by multiple elimination. *ACM Trans. Math. Software*, 11(2):141–153, 1985.

[22] MPI: A Message Passing Interface Standard, version 1.1, jun 1995. Available from `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`.

[23] F. Pellegrini. Static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proc. SHPCC'94, Knoxville*, pages 486–493. IEEE, May 1994.

[24] F. Pellegrini. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In *Proc. EuroPar, Rennes*, LNCS 4641, pages 191–200, August 2007.

[25] F. Pellegrini. Scotch 5.0 User's Guide. Technical report, LaBRI, Université Bordeaux I, August 2007. Available from `http://www.labri.fr/~pelegrin/scotch/`.

[26] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *Proc. HPCN'96, Brussels*, LNCS 1067, pages 493–498, April 1996.

[27] F. Pellegrini and J. Roman. Sparse matrix ordering with scotch. In *Proc. HPCN'97, Vienna*, LNCS 1225, pages 370–378, April 1997.

[28] F. Pellegrini, J. Roman, and P. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. *Concurrency: Practice and Experience*, 12:69–84, 2000.

[29] R. Schreiber. Scalability of sparse direct solvers. Technical Report TR 92.13, RIACS, NASA Ames Research Center, May 1992.

[30] W. F. Tinney and J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *J. Proc. IEEE*, 55:1801–1809, 1967.