

PT-Scotch and libPTScotch 7.0 User's Guide

(version 7.0.18)

François Pellegrini
Université de Bordeaux & LaBRI, UMR CNRS 5800
TadAAM team, INRIA Bordeaux Sud-Ouest
351 cours de la Libération, 33405 TALENCE, FRANCE
pelegrin@labri.fr

October 31, 2026

Abstract

This document describes the capabilities and operations of PT-Scotch and libScotch, a software package and a software library which compute parallel static mappings and parallel sparse matrix block orderings of distributed 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 **5**
- 1.1 Static mapping . . . 5
- 1.2 Sparse matrix ordering . . . 6
- 1.3 Contents of this document . . . 6

2 The Scotch project **6**
- 2.1 Description . . . 6
- 2.2 Availability . . . 7

3 Algorithms **7**
- 3.1 Parallel static mapping by Dual Recursive Bipartitioning . . . 7
  - 3.1.1 Static mapping . . . 7
  - 3.1.2 Cost function and performance criteria . . . 8
  - 3.1.3 The Dual Recursive Bipartitioning algorithm . . . 9
  - 3.1.4 Partial cost function . . . 10
  - 3.1.5 Parallel graph bipartitioning methods . . . 10
  - 3.1.6 Mapping onto variable-sized architectures . . . 12
- 3.2 Parallel sparse matrix ordering by hybrid incomplete nested dissection 12
  - 3.2.1 Hybrid incomplete nested dissection . . . 13
  - 3.2.2 Parallel ordering . . . 13
  - 3.2.3 Performance criteria . . . 17

4 Updates **17**
- 4.1 Changes from version 6.0 . . . 17
- 4.2 Changes from version 5.0 . . . 18
- 4.3 Changes from version 5.1 . . . 18
- 4.4 Changes from version 6.1 . . . 18

5 Files and data structures **19**
- 5.1 Distributed graph files . . . 19
- 5.2 Auxiliary distributed mesh files . . . 20

6 Programs **21**
- 6.1 Invocation . . . 21
- 6.2 File names . . . 22
  - 6.2.1 Sequential and parallel file opening . . . 22
- 6.3 Using multi-threading . . . 22
  - 6.3.1 Using compressed files . . . 23
- 6.4 Description . . . 23
  - 6.4.1 `adm2dgr` . . . 23
  - 6.4.2 `dgmap` / `dgpart` . . . 24
  - 6.4.3 `dgord` . . . 26
  - 6.4.4 `dgpart` . . . 27
  - 6.4.5 `dgscat` / `gscat` . . . 28
  - 6.4.6 `dgtst` . . . 28

7 Library **29**
7.1 Using multi-threading . . . 30
7.1.1 Compiling at proper thread level . . . 30
7.1.2 Running at proper thread level . . . 30
7.1.3 Launching with threads . . . 30
7.2 Calling the routines of libScotch . . . 31
7.2.1 Calling from C . . . 31
7.2.2 Calling from Fortran . . . 31
7.2.3 Compiling and linking . . . 33
7.2.4 Machine word size issues . . . 33
7.3 Data types . . . 34
7.3.1 `SCOTCH_Dgraph` distributed graph type . . . 34
7.3.2 `SCOTCH_Dmesh` distributed mesh type . . . 39
7.3.3 `SCOTCH_Dordering` distributed block ordering type . . . 42
7.4 Strategy strings . . . 43
7.4.1 Using default strategy strings . . . 43
7.4.2 Parallel mapping strategy strings . . . 44
7.4.3 Parallel graph bipartitioning strategy strings . . . 45
7.4.4 Parallel ordering strategy strings . . . 48
7.4.5 Parallel node separation strategy strings . . . 50
7.5 Distributed graph handling routines . . . 53
7.5.1 `SCOTCH_dgraphAlloc` . . . 53
7.5.2 `SCOTCH_dgraphBand` . . . 53
7.5.3 `SCOTCH_dgraphBuild` . . . 54
7.5.4 `SCOTCH_dgraphBuildGrid3D` . . . 56
7.5.5 `SCOTCH_dgraphCheck` . . . 57
7.5.6 `SCOTCH_dgraphCoarsen` . . . 58
7.5.7 `SCOTCH_dgraphCoarsenVertLocMax` . . . 59
7.5.8 `SCOTCH_dgraphData` . . . 60
7.5.9 `SCOTCH_dgraphExit` . . . 62
7.5.10 `SCOTCH_dgraphFree` . . . 62
7.5.11 `SCOTCH_dgraphGather` . . . 63
7.5.12 `SCOTCH_dgraphGrow` . . . 63
7.5.13 `SCOTCH_dgraphInducePart` . . . 64
7.5.14 `SCOTCH_dgraphInit` . . . 66
7.5.15 `SCOTCH_dgraphRedist` . . . 66
7.5.16 `SCOTCH_dgraphScatter` . . . 67
7.5.17 `SCOTCH_dgraphSize` . . . 68
7.5.18 `SCOTCH_dgraphSizeof` . . . 69
7.5.19 `SCOTCH_dgraphStat` . . . 69
7.6 Distributed graph I/O routines . . . 70
7.6.1 `SCOTCH_dgraphLoad` . . . 70
7.6.2 `SCOTCH_dgraphSave` . . . 71
7.7 Data handling and exchange routines . . . 72
7.7.1 `SCOTCH_dgraphGhst` . . . 72
7.7.2 `SCOTCH_dgraphHalo` . . . 72
7.7.3 `SCOTCH_dgraphHaloAsync` . . . 73
7.7.4 `SCOTCH_dgraphHaloWait` . . . 74
7.8 Distributed graph mapping and partitioning routines . . . 75
7.8.1 `SCOTCH_dgraphMap` . . . 75
7.8.2 `SCOTCH_dgraphMapCompute` . . . 76

7.8.3 SCOTCH_dgraphMapExit . . . 76
7.8.4 SCOTCH_dgraphMapInit . . . 77
7.8.5 SCOTCH_dgraphMapSave . . . 77
7.8.6 SCOTCH_dgraphMapStat . . . 78
7.8.7 SCOTCH_dgraphMapView . . . 80
7.8.8 SCOTCH_dgraphPart . . . 80
7.9 Distributed graph ordering routines . . . 81
7.9.1 SCOTCH_dgraphOrderCblkDist . . . 81
7.9.2 SCOTCH_dgraphOrderCompute . . . 82
7.9.3 SCOTCH_dgraphOrderComputeList . . . 82
7.9.4 SCOTCH_dgraphOrderExit . . . 83
7.9.5 SCOTCH_dgraphOrderInit . . . 83
7.9.6 SCOTCH_dgraphOrderSave . . . 84
7.9.7 SCOTCH_dgraphOrderSaveMap . . . 84
7.9.8 SCOTCH_dgraphOrderSaveTree . . . 85
7.9.9 SCOTCH_dgraphOrderPerm . . . 86
7.9.10 SCOTCH_dgraphOrderTreeDist . . . 86
7.10 Centralized ordering handling routines . . . 87
7.10.1 SCOTCH_dgraphCorderExit . . . 87
7.10.2 SCOTCH_dgraphCorderInit . . . 88
7.10.3 SCOTCH_dgraphOrderGather . . . 89
7.11 Distributed mesh handling routines . . . 89
7.11.1 SCOTCH_dmeshAlloc . . . 89
7.11.2 SCOTCH_dmeshBuildAdm . . . 90
7.11.3 SCOTCH_dmeshData . . . 91
7.11.4 SCOTCH_dmeshDgraphDual . . . 92
7.11.5 SCOTCH_dmeshExit . . . 93
7.11.6 SCOTCH_dmeshFree . . . 93
7.11.7 SCOTCH_dmeshInit . . . 94
7.11.8 SCOTCH_dmeshSize . . . 94
7.11.9 SCOTCH_dmeshSizeof . . . 95
7.12 Distributed mesh I/O routines . . . 95
7.12.1 SCOTCH_dmeshLoad . . . 95
7.13 Strategy handling routines . . . 96
7.13.1 SCOTCH_stratExit . . . 96
7.13.2 SCOTCH_stratInit . . . 97
7.13.3 SCOTCH_stratSave . . . 97
7.14 Strategy creation routines . . . 97
7.14.1 SCOTCH_stratDgraphClusterBuild . . . 98
7.14.2 SCOTCH_stratDgraphMap . . . 99
7.14.3 SCOTCH_stratDgraphMapBuild . . . 99
7.14.4 SCOTCH_stratDgraphOrder . . . 100
7.14.5 SCOTCH_stratDgraphOrderBuild . . . 100
7.15 Other data structure routines . . . 101
7.15.1 SCOTCH_dmapAlloc . . . 101
7.15.2 SCOTCH_dmapSizeof . . . 101
7.15.3 SCOTCH_dorderAlloc . . . 102
7.15.4 SCOTCH_dorderSizeof . . . 102
7.16 Error handling routines . . . 102
7.16.1 SCOTCH_errorPrint . . . 103
7.16.2 SCOTCH_errorPrintW . . . 103

7.16.3 SCOTCH_errorProg . . . 103
7.17 Random generator handling . . . 103
7.17.1 SCOTCH_randomProc . . . 104
7.17.2 SCOTCH_randomReset . . . 104
7.17.3 SCOTCH_randomSeed . . . 105
7.17.4 SCOTCH_randomVal . . . 105
7.18 Context handling routines . . . 105
7.18.1 SCOTCH_contextBindDgraph . . . 106
7.19 Memory management . . . 106
7.19.1 SCOTCH_memCur . . . 106
7.19.2 SCOTCH_memMax . . . 107
7.20 PARMETIS compatibility library . . . 108
7.20.1 ParMETIS_V3_Mesh2Dual . . . 108
7.20.2 ParMETIS_V3_NodeND . . . 109
7.20.3 ParMETIS_V3_PartGeomKway . . . 109
7.20.4 ParMETIS_V3_PartKway . . . 110

8 Installation **111**
8.1 Thread issues . . . 112
8.2 File compression issues . . . 112
8.3 Machine word size issues . . . 112

9 Examples **113**

1 Introduction

1.1 Static mapping

The efficient execution of a parallel program on a parallel machine requires that the communicating processes of the program be assigned to the processors of the machine so as to minimize its overall running time. When processes have a limited duration and their logical dependencies are accounted for, this optimization problem is referred to as scheduling. When processes are assumed to coexist simultaneously for the entire duration of the program, it is referred to as mapping. It amounts to balancing the computational weight of the processes among the processors of the machine, while reducing the cost of communication by keeping intensively intercommunicating processes on nearby processors.

In most cases, the underlying computational structure of the parallel programs to map can be conveniently modeled as a graph in which vertices correspond to processes that handle distributed pieces of data, and edges reflect data dependencies. The mapping problem can then be addressed by assigning processor labels to the vertices of the graph, so that all processes assigned to some processor are loaded and run on it. In a SPMD context, this is equivalent to the distribution across processors of the data structures of parallel programs; in this case, all pieces of data assigned to some processor are handled by a single process located on this processor.

A mapping is called static if it is computed prior to the execution of the program. Static mapping is NP-complete in the general case [10]. Therefore, many studies have been carried out in order to find sub-optimal solutions in reasonable time, including the development of specific algorithms for common topologies such as the hypercube [8, 16]. When the target machine is assumed to have a communication network in the shape of a complete graph, the static mapping problem turns into the

partitioning problem, which has also been intensely studied [3, 17, 25, 26, 40]. However, when mapping onto parallel machines the communication network of which is not a bus, not accounting for the topology of the target machine usually leads to worse running times, because simple cut minimization can induce more expensive long-distance communication [16, 43]; the static mapping problem is gaining popularity as most of the newer massively parallel machines have a strongly NUMA architecture

1.2 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.

1.3 Contents of this document

This document describes the capabilities and operations of PT-Scotch, a software package devoted to parallel static mapping and 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* [35].

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 5 defines the formats of the files used in PT-Scotch, section 6 describes the programs of the PT-Scotch distribution, and section 7 defines the interface and operations of the parallel routines of the libScotch library. Section 8 explains how to obtain and install the Scotch distribution. Finally, some practical examples are given in section 9.

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 Bacchus project of INRIA Bordeaux Sud-Ouest. 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 [33], all of which have been implemented in

the Scotch software package [37]. 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 [38, 39]. 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, 34]. Version 5.0 of Scotch was the first one to comprise parallel graph ordering routines [7], and version 5.1 started offering parallel graph partitioning features, while parallel static mapping will be available in the next release.

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 7.0 is distributed is the CeCILL-C license [4], which has basically the same features as the GNU LGPL ("*Lesser General Public License*") [29]: 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 8 to see how to obtain the free/libre distribution of Scotch.

3 Algorithms

3.1 Parallel static mapping by Dual Recursive Bipartitioning

For a detailed description of the sequential implementation of this mapping algorithm and an extensive analysis of its performance, please refer to [33, 36]. In the next sections, we will only outline the most important aspects of the algorithm.

3.1.1 Static mapping

The parallel program to be mapped onto the target architecture is modeled by a valuated unoriented graph S called source graph or process graph, the vertices of which represent the processes of the parallel program, and the edges of which the communication channels between communicating processes. Vertex- and edge- valuations associate with every vertex v_S and every edge e_S of S integer numbers $w_S(v_S)$ and $w_S(e_S)$ which estimate the computation weight of the corresponding process and the amount of communication to be transmitted on the channel, respectively.

The target machine onto which is mapped the parallel program is also modeled by a valuated unoriented graph T called target graph or architecture graph. Vertices v_T and edges e_T of T are assigned integer weights $w_T(v_T)$ and $w_T(e_T)$, which estimate the computational power of the corresponding processor and the cost of traversal of the inter-processor link, respectively.

A mapping from S to T consists of two applications $\tau_{S,T} : V(S) \longrightarrow V(T)$ and $\rho_{S,T} : E(S) \longrightarrow \mathcal{P}(E(T))$, where $\mathcal{P}(E(T))$ denotes the set of all simple loopless paths which can be built from $E(T)$. $\tau_{S,T}(v_S) = v_T$ if process v_S of S is mapped onto processor v_T of T, and $\rho_{S,T}(e_S) = \{e_T^1, e_T^2, \ldots, e_T^n\}$ if communication channel e_S of S is routed through communication links e_T^1, e_T^2, ..., e_T^n of T. $|\rho_{S,T}(e_S)|$ denotes the dilation of edge e_S, that is, the number of edges of $E(T)$ used to route e_S.

3.1.2 Cost function and performance criteria

The computation of efficient static mappings requires an *a priori* knowledge of the dynamic behavior of the target machine with respect to the programs which are run on it. This knowledge is synthesized in a cost function, the nature of which determines the characteristics of the desired optimal mappings. The goal of our mapping algorithm is to minimize some communication cost function, while keeping the load balance within a specified tolerance. The communication cost function f_C that we have chosen is the sum, for all edges, of their dilation multiplied by their weight:

$$f_C(\tau_{S,T}, \rho_{S,T}) \stackrel{\text{def}}{=} \sum_{e_S \in E(S)} w_S(e_S) \, |\rho_{S,T}(e_S)| \ .$$

This function, which has already been considered by several authors for hypercube target topologies [8, 16, 20], has several interesting properties: it is easy to compute, allows incremental updates performed by iterative algorithms, and its minimization favors the mapping of intensively intercommunicating processes onto nearby processors; regardless of the type of routage implemented on the target machine (store-and-forward or cut-through), it models the traffic on the interconnection network and thus the risk of congestion.

The strong positive correlation between values of this function and effective execution times has been experimentally verified by Hammond [16] on the CM-2, and by Hendrickson and Leland [21] on the nCUBE 2.

The quality of mappings is evaluated with respect to the criteria for quality that we have chosen: the balance of the computation load across processors, and the minimization of the interprocessor communication cost modeled by function f_C. These criteria lead to the definition of several parameters, which are described below.

For load balance, one can define μ_{map}, the average load per computational power unit (which does not depend on the mapping), and δ_{map}, the load imbalance ratio, as

$$\mu_{map} \stackrel{\text{def}}{=} \frac{\displaystyle\sum_{v_S \in V(S)} w_S(v_S)}{\displaystyle\sum_{v_T \in V(T)} w_T(v_T)} \qquad \text{and}$$

$$\delta_{map} \stackrel{\text{def}}{=} \frac{\displaystyle\sum_{v_T \in V(T)} \left| \left(\frac{1}{w_T(v_T)} \sum_{\substack{v_S \in V(S) \\ \tau_{S,T}(v_S) = v_T}} w_S(v_S) \right) - \mu_{map} \right|}{\displaystyle\sum_{v_S \in V(S)} w_S(v_S)} \ .$$

However, since the maximum load imbalance ratio is provided by the user in input of the mapping, the information given by these parameters is of little interest, since

what matters is the minimization of the communication cost function under this load balance constraint.

For communication, the straightforward parameter to consider is f_C. It can be normalized as μ_{exp}, the average edge expansion, which can be compared to μ_{dil}, the average edge dilation; these are defined as

$$\mu_{exp} \stackrel{\text{def}}{=} \frac{f_C}{\sum_{e_S \in E(S)} w_S(e_S)} \qquad \text{and} \qquad \mu_{dil} \stackrel{\text{def}}{=} \frac{\sum_{e_S \in E(S)} |\rho_{S,T}(e_S)|}{|E(S)|} \ .$$

$\delta_{exp} \stackrel{\text{def}}{=} \frac{\mu_{exp}}{\mu_{dil}}$ is smaller than 1 when the mapper succeeds in putting heavily intercommunicating processes closer to each other than it does for lightly communicating processes; they are equal if all edges have same weight.

3.1.3 The Dual Recursive Bipartitioning algorithm

Our mapping algorithm uses a divide and conquer approach to recursively allocate subsets of processes to subsets of processors [33].

It starts by considering a set of processors, also called domain, containing all the processors of the target machine, and with which is associated the set of all the processes to map. At each step, the algorithm bipartitions a yet unprocessed domain into two disjoint subdomains, and calls a graph bipartitioning algorithm to split the subset of processes associated with the domain across the two subdomains, as sketched in the following.

```
mapping (D, P)
Set_Of_Processors  D;
Set_Of_Processes   P;
{
  Set_Of_Processors  D0, D1;
  Set_Of_Processes   P0, P1;

  if (|P| == 0) return;  /* If nothing to do.     */
  if (|D| == 1) {        /* If one processor in D */
    result (D, P);       /* P is mapped onto it.  */
    return;
  }

  (D0, D1) = processor_bipartition (D);
  (P0, P1) = process_bipartition   (P, D0, D1);
  mapping (D0, P0);      /* Perform recursion. */
  mapping (D1, P1);
}
```

The association of a subdomain with every process defines a partial mapping of the process graph. As bipartitionings are performed, the subdomain sizes decrease, up to give a complete mapping when all subdomains are of size one.

The above algorithm lies on the ability to define five main objects:

- a domain structure, which represents a set of processors in the target architecture;
- a domain bipartitioning function, which, given a domain, bipartitions it in two disjoint subdomains;
- a domain distance function, which gives, in the target graph, a measure of the distance between two disjoint domains. Since domains may not be convex nor connected, this distance may be estimated. However, it must respect certain homogeneity properties, such as giving more accurate results as domain sizes

decrease. The domain distance function is used by the graph bipartitioning algorithms to compute the communication function to minimize, since it allows the mapper to estimate the dilation of the edges that link vertices which belong to different domains. Using such a distance function amounts to considering that all routings will use shortest paths on the target architecture, which is how most parallel machines actually do. We have thus chosen that our program would not provide routings for the communication channels, leaving their handling to the communication system of the target machine;

- a process subgraph structure, which represents the subgraph induced by a subset of the vertex set of the original source graph;
- a process subgraph bipartitioning function, which bipartitions subgraphs in two disjoint pieces to be mapped onto the two subdomains computed by the domain bipartitioning function.

All these routines are seen as black boxes by the mapping program, which can thus accept any kind of target architecture and process bipartitioning functions.

3.1.4 Partial cost function

The production of efficient complete mappings requires that all graph bipartitionings favor the criteria that we have chosen. Therefore, the bipartitioning of a subgraph S' of S should maintain load balance within the user-specified tolerance, and minimize the partial communication cost function f'_C, defined as

$$f'_C(\tau_{S,T}, \rho_{S,T}) \stackrel{\text{def}}{=} \sum_{\substack{v \in V(S') \\ \{v,v'\} \in E(S)}} w_S(\{v,v'\}) \, |\rho_{S,T}(\{v,v'\})| \ ,$$

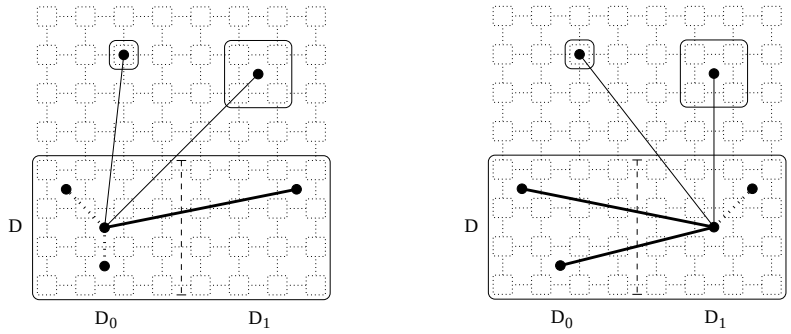
which accounts for the dilation of edges internal to subgraph S' as well as for the one of edges which belong to the cocycle of S', as shown in Figure 1. Taking into account the partial mapping results issued by previous bipartitionings makes it possible to avoid local choices that might prove globally bad, as explained below. This amounts to incorporating additional constraints to the standard graph bipartitioning problem, turning it into a more general optimization problem termed as skewed graph partitioning by some authors [23].

3.1.5 Parallel graph bipartitioning methods

The core of our parallel recursive mapping algorithm uses process graph parallel bipartitioning methods as black boxes. It allows the mapper to run any type of graph bipartitioning method compatible with our criteria for quality. Bipartitioning jobs maintain an internal image of the current bipartition, indicating for every vertex of the job whether it is currently assigned to the first or to the second subdomain. It is therefore possible to apply several different methods in sequence, each one starting from the result of the previous one, and to select the methods with respect to the job characteristics, thus enabling us to define mapping strategies. The currently implemented graph bipartitioning methods are listed below.

Band

Like the multi-level method which will be described below, the band method is a meta-algorithm, in the sense that it does not itself compute partitions, but rather helps other partitioning algorithms perform better. It is a refinement



a. Initial position. **b.** After one vertex is moved.

Figure 1: Edges accounted for in the partial communication cost function when bipartitioning the subgraph associated with domain D between the two subdomains D_0 and D_1 of D. Dotted edges are of dilation zero, their two ends being mapped onto the same subdomain. Thin edges are cocycle edges.

algorithm which, from a given initial partition, extracts a band graph of given width (which only contains graph vertices that are at most at this distance from the separator), calls a partitioning strategy on this band graph, and prolongs[1] back the refined partition on the original graph. This method was designed to be able to use expensive partitioning heuristics, such as genetic algorithms, on large graphs, as it dramatically reduces the problem space by several orders of magnitude. However, it was found that, in a multi-level context, it also improves partition quality, by coercing partitions in a problem space that derives from the one which was globally defined at the coarsest level, thus preventing local optimization refinement algorithms to be trapped in local optima of the finer graphs [6].

Diffusion

This global optimization method, the sequential formulation of which is presented in [34], flows two kinds of antagonistic liquids, scotch and anti-scotch, from two source vertices, and sets the new frontier as the limit between vertices which contain scotch and the ones which contain anti-scotch. In order to add load-balancing constraints to the algorithm, a constant amount of liquid disappears from every vertex per unit of time, so that no domain can spread across more than half of the vertices. Because selecting the source vertices is essential to the obtainment of useful results, this method has been hard-coded so that the two source vertices are the two vertices of highest indices, since in the band method these are the anchor vertices which represent all of the removed vertices of each part. Therefore, this method must be used on band graphs only, or on specifically crafted graphs.

Multi-level

This algorithm, which has been studied by several authors [3, 18, 25] and should be considered as a strategy rather than as a method since it uses other methods as parameters, repeatedly reduces the size of the graph to bipartition by finding matchings that collapse vertices and edges, computes a partition for the coarsest graph obtained, and prolongs the result back to the original graph,

[1] While a *projection* is an application to a space of lower dimension, a *prolongation* refers to an application to a space of higher dimension. Yet, the term projection is also commonly used to refer to such a propagation, most often in the context of a multilevel framework.

as shown in Figure 2. The multi-level method, when used in conjunction with

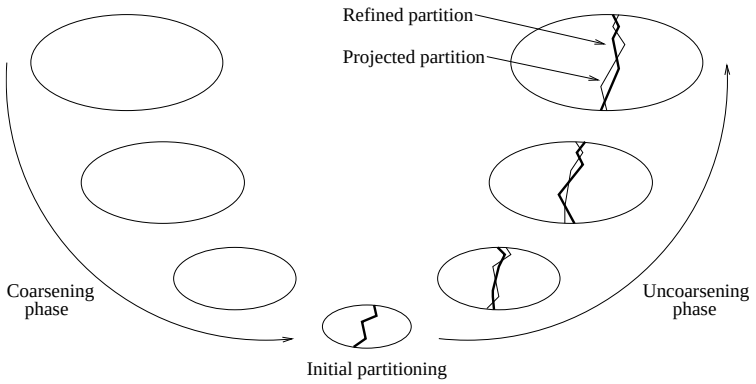


Figure 2: The multi-level partitioning process. In the uncoarsening phase, the light and bold lines represent for each level the prolonged partition obtained from the coarser graph, and the partition obtained after refinement, respectively.

the banded diffusion method to refine the prolonged partitions at every level, usually stabilizes quality irrespective of the number of processors which run the parallel static mapper.

3.1.6 Mapping onto variable-sized architectures

Several constrained graph partitioning problems can be modeled as mapping the problem graph onto a target architecture, the number of vertices and topology of which depend dynamically on the structure of the subgraphs to bipartition at each step.

Variable-sized architectures are supported by the DRB algorithm in the following way: at the end of each bipartitioning step, if any of the variable subdomains is empty (that is, all vertices of the subgraph are mapped only to one of the subdomains), then the DRB process stops for both subdomains, and all of the vertices are assigned to their parent subdomain; else, if a variable subdomain has only one vertex mapped onto it, the DRB process stops for this subdomain, and the vertex is assigned to it.

The moment when to stop the DRB process for a specific subgraph can be controlled by defining a bipartitioning strategy that tests for the validity of a criterion at each bipartitioning step, and maps all of the subgraph vertices to one of the subdomains when it becomes false.

3.2 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).

3.2.1 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 [42] 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, 13, 31]). However, the algorithm is intrinsically sequential, and very little can be theoretically proved about its efficiency.

The nested dissection algorithm [14] 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, 30], 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 [15, 25, 38] to the ones obtained with the minimum degree algorithm [31]. 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, 11, 12, 15, 38, 41, and included references].

Due to their complementary nature, several schemes have been proposed to hybridize the two methods [24, 27, 38]. 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 [39].

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.

3.2.2 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 3: once a separator

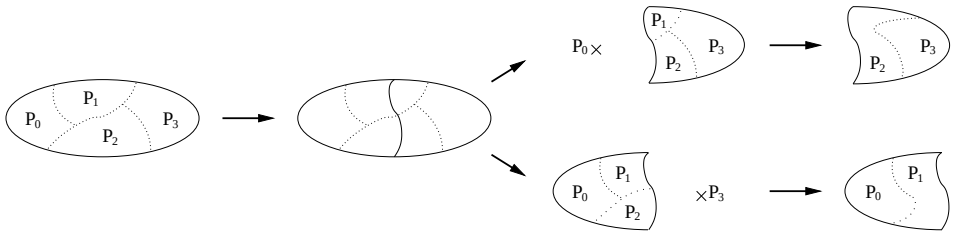


Figure 3: Diagram of a nested dissection step for a (sub-)graph distributed across four processors. Once the separator is known, the two induced subgraphs are built and folded (this can be done in parallel for both subgraphs), yielding two subgraphs, each of them distributed across two processors.

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 [39], 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, 22, 27]. 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 prolonging back this separator, from coarser to finer graphs, up to the original graph. Most often, a local optimization algorithm, such as Kernighan-Lin [28] or Fiduccia-Mattheyses [9] (FM), is used in the uncoarsening phase to refine the partition that

is prolonged 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 4. 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 4, 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 4, 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 4: the routines of the sequential Scotch library are used on every processor to complete the coarsening process, compute an initial partition, and prolong it back up to the largest centralized coarsened graph stored on the processor. Then, the partitions are prolonged back in parallel to the finer distributed graphs, selecting the best partition between the two available when prolonging to a level where fold-dup had been performed. This distributed prolongation 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 prolonged 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 prolonged separator, local refinement algorithms need only to be passed a subgraph that contains the vertices that are very close to

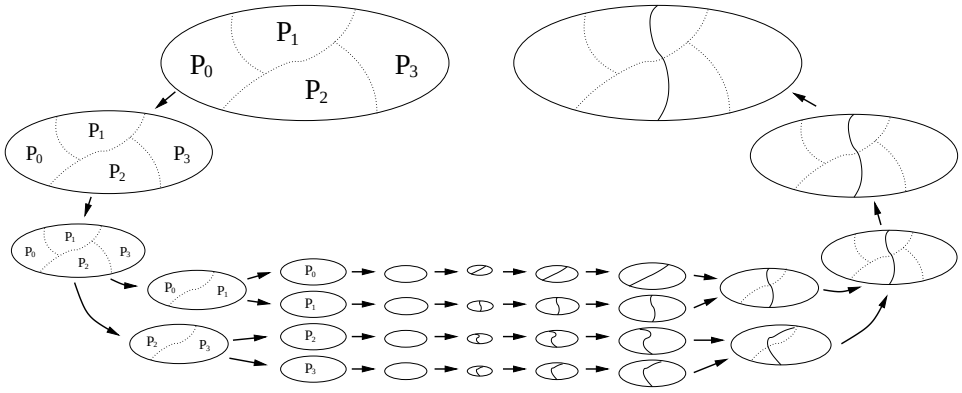


Figure 4: Diagram of the parallel computation of the separator of a graph distributed across four processors, by parallel coarsening with folding-with-duplication in the last stages, multi-sequential computation of initial partitions that are locally prolonged back and refined on every processor, and then parallel uncoarsening of the best partition encountered.

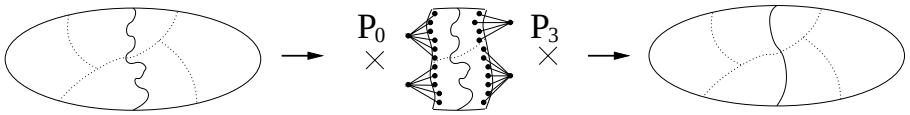


Figure 5: Creation of a distributed band graph. Only vertices closest to the separator are kept. Other vertices are replaced by anchor vertices of equivalent total weight, linked to band vertices of the last layer. There are two anchor vertices per processor, to reduce communication. Once the separator has been refined on the band graph using some local optimization algorithm, the new separator is prolonged back to the original distributed graph.

the prolonged separator.

The computation and use of distributed band graphs is outlined in Figure 5. 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 prolonged back to the original distributed graph.

Basing on these band graphs, we have implemented a multi-sequential refinement algorithm, outlined in Figure 6. At every distributed uncoarsening step, a distributed band graph is created. Centralized copies of this band graph are then 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

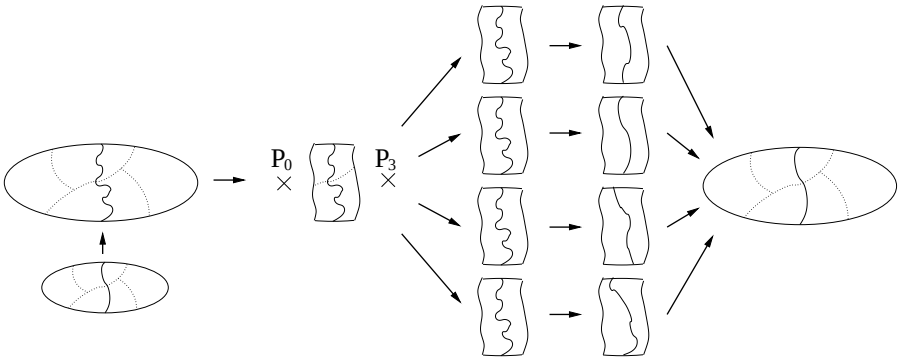


Figure 6: Diagram of the multi-sequential refinement of a separator prolonged back from a coarser graph distributed across four processors to its finer distributed graph. Once the distributed band graph is built from the finer graph, a centralized version of it is gathered on every participating processor. 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.

band separator is prolonged back to the distributed graph, and the uncoarsening process goes on.

3.2.3 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[2], 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 Updates

4.1 Changes from version 6.0

Thread management in Scotch is now dynamic. This allows the user to control dynamically the number of threads that are used by the threaded algorithms of the libPTScotch and libScotch libraries and, consequently, by the PT-Scotch standalone programs that call them. Users can control concurrency in the libPTScotch by using `SCOTCH_Context` objects. These objects define user-

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

configurable execution contexts, in which libPTScotch library routines can be executed independently from others.

Execution contexts also comprise option values, which dynamically determine the behavior of the Scotch routines. In previous versions, such behavior was controlled at compile-time by flags `COMMON_RANDOM_FIXED_SEED` and `SCOTCH_DETERMINISTIC`. It is now possible to define these flags dynamically, using the relevant option setting routine. Please refer to the Scotch user's manual for a comprehensive description of `SCOTCH_Context` features.

Thanks to this new model, PT-Scotch can use multi-threaded algorithms when several threads are available per MPI node. Two levels of multi-threading are available: multi-threading with a single thread handling MPI communications, and multi-threading with several threads being able to perform MPI communication simultaneously. This behavior is controlled at compile-time by flags `SCOTCH_PTHREAD` and `SCOTCH_PTHREAD_MPI`. Please refer to section 7.1 for more information.

Support for CMake has been added.

4.2 Changes from version 5.0

PT-Scotch now provides routines to compute in parallel partitions of distributed graphs.

A new integer index type has been created in the Fortran interface, to address array indices larger than the maximum value which can be stored in a regular integer. Please refer to Section 8.3 for more information.

A new set of routines has been designed, to ease the use of the libScotch as a dynamic library. The `SCOTCH_version` routine returns the version, release and patchlevel numbers of the library being used. The `SCOTCH_*Alloc` routines, which are only available in the C interface at the time being, dynamically allocate storage space for the opaque API Scotch structures, which frees application programs from the need to be systematically recompiled because of possible changes of Scotch structure sizes.

4.3 Changes from version 5.1

Unlike its sequential counterpart, version 6.0 of PT-Scotch does not bring major algorithmic improvements with respect to the latest 5.1.12 release of the 5.1 branch.

In order to ease the work of people writing numerical solvers, it exposes in its interface a new distributed graph handling routine, `SCOTCH_dgraphRedist`, that builds a redistributed graph from an existing distributed graph and partition data. See Section 7.5.15.

4.4 Changes from version 6.1

The prototype `Dmesh` distributed mesh data type has been created, to allow one to compute dual distributed graphs, using the `dmeshDgraphDual()` routine (see Section 7.11.4). This routine, and others, allowed for the addition of the `ParMETIS_V3_Mesh2Dual()` routine to the ParMeTiS compatibility module (see Section 7.20.1).

5 Files and data structures

For the sake of portability and readability, 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.

5.1 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* [35]). 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 9, page 37, for a detailed example), followed by the number of this file in the sequence (ranging from 0 to (`procglbnbr` − 1), and analogous to `proclocnum` in Figure 9). 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 `vertlocnbr`), followed by its local number of arcs (analogous to `edgelocnbr`). The fifth line holds two figures: the graph base index value (`baseval`) 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 [19], 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 7 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 7: Two complementary distributed graph files representing a cube distributed across two processors.

5.2 Auxiliary distributed mesh files

The auxiliary distributed mesh file format has been created to support the implementation of the prototype `Dmesh` class representing distributed meshes. The main use of these files, which usually end in "`.adm`", is as input of the `dmeshLoad()` routine. Unlike the SCOTCH mesh data structures and file formats, which are symmetric, this mesh format only provides adjacency lists for elements: the list of nodes connected to a given element is provided, while the opposite is not, hence the name "auxiliary".

The first line of an auxiliary distributed mesh file holds the distributed mesh file version number, which is currently `3`. The second line holds the number of files across which the mesh data is distributed (referred to as `procglbnbr` in LIBSCOTCH; see for instance Figure 9, page 37, for a detailed example in the case of distributed graphs), followed by the number of this file in the sequence (ranging from 0 to (`procglbnbr` − 1), and analogous to `proclocnum` in Figure 9). The third line holds the global number of mesh element vertices (referred to as `velmglbnbr`), followed by the global number of node vertices (referred to as `vnodglbnbr`). The fourth line holds the number of elements contained in this mesh fragment (`velmlocnbr`), followed by the sum of all neighboring node indices for these local elements (`eelm locnbr`). The fifth line holds two figures: the mesh base index value (`baseval`) and a numeric flag.

The mesh base index value records the value of the starting index used to describe the mesh; it is usually 0 when the mesh has been output by C programs, and

1 for Fortran programs. Its purpose is to ease the manipulation of meshes within each of these two environments, while providing compatibility between them. The numeric flag, similar to the one used by the Chaco graph format [19], is made of three decimal digits; it is currently not used.

This header data is then followed by as many lines as there are elements in the mesh fragment, that is, `velmlocnbr` lines. Each of these lines begins with the number of vertices contained in the element described by that line.

In the case of the mesh distributed on three processes represented in Figure 11, the contents of the three corresponding auxiliary distributed mesh files are shown in Figure 8.

```
3
3 0
6 12
3 12
1 000
4 5 1 7 11
4 8 11 6 7
4 4 1 2 7
```

```
3
3 1
6 12
1 4
1 000
4 10 6 3 8
```

```
3
3 2
6 12
2 8
1 000
4 12 7 2 6
4 3 9 12 6
```

Figure 8: The three auxiliary distributed mesh files corresponding to the mesh distributed across three processors of Figure 11, page 41.

6 Programs

6.1 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 9.

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 5, and use the `dgtst` program of the PT-Scotch distribution to check the consistency of your 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.

6.2 File names

6.2.1 Sequential and parallel file opening

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 5.1.

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.

6.3 Using multi-threading

Starting from version 6.1.0, thread management in Scotch is dynamic. This allows the user to control dynamically the number of threads that are used by the threaded algorithms of the libScotch library and, consequently, by the Scotch standalone programs that call them. These algorithms are enabled when Scotch is compiled with the flag "`-DSCOTCH_PTHREAD`" set.

The behavior of the libScotch and all the executables that rely on it (including the Scotch standalone programs themselves) can be modified dynamically by way of environment variables, which take precedence over the compilation flags with same purpose.

- SCOTCH_PTHREAD_NUMBER: this environment variable sets the prescribed maximum number of threads that Scotch may use in the course of its computations. A value of -1 indicates to use as many threads as provided by the system at launch time. Setting this number to 1 will coerce Scotch into using only purely sequential algorithms (which may differ in nature from their multi-threaded counterparts).
- SCOTCH_DETERMINISTIC: when set to 0, faster, non-deterministic, multi-threaded algorithms may be used; when set to 1, fully deterministic, albeit sometimes slower, multi-threaded algorithms are always used.
- SCOTCH_RANDOM_FIXED_SEED: when set to 1, the same pseudo-random seed is used for each run, allowing for reproducibility in case only deterministic algorithms are used; when set to 0, a new pseudo-random seed is used for each run.

6.3.1 Using compressed files

Starting from version 5.0.6, Scotch allows users to provide and retrieve data in compressed form. Since this feature requires that the compression and decompression tasks run in the same time as data is read or written, it can only be done on systems which support multi-threading (Posix threads) or multi-processing (by means of `fork` system calls).

To determine if a stream has to be handled in compressed form, Scotch checks its extension. If it is "`.gz`" (`gzip` format), "`.bz2`" (`bzip2` format) or "`.lzma`" (`lzma` format), the stream is assumed to be compressed according to the corresponding format. A filter task will then be used to process it accordingly if the format is implemented in Scotch and enabled on your system.

To date, data can be read and written in `bzip2` and `gzip` formats, and can also be read in the `lzma` format. Since the compression ratio of `lzma` on Scotch graphs is 30% better than the one of `gzip` and `bzip2` (which are almost equivalent in this case), the `lzma` format is a very good choice for handling very large graphs. To see how to enable compressed data handling in Scotch, please refer to Section 8.

When the compressed format allows it, several files can be provided on the same stream, and be uncompressed on the fly. For instance, the command "`cat brol.grf.gz brol.xyz.gz | gout -.gz -.gz -Mn - brol.iv`" concatenates the topology and geometry data of some graph `brol` and feed them as a single compressed stream to the standard input of program `gout`, hence the "`-.gz`" to indicate a compressed standard stream.

6.4 Description

6.4.1 `adm2dgr`

Synopsis

`adm2dgr` [*input_mesh_file* [*output_graph_file*]] *options*

Description

The `adm2dgr` program is not a standard Scotch program. It is part of the libScotchMeTiS module, and is only available when the latter is installed.

Its main use is to test the `ParMETIS_V3_Mesh2dual()` routine of the LIBSCOTCHMETIS library (see Section 7.20.1).

The `adm2dgr` program computes in parallel the dual graph (that is, the element graph) of a distributed mesh provided as an auxiliary distributed mesh file. The output distributed graph is stored in the form of a SCOTCH distributed graph file.

The adjacency relationship between two elements in the dual graph is defined by the number of nodes they have in common. This number can be provided on the command line.

The *input_mesh_file* filename refers to an auxiliary distributed mesh, according to the semantics defined in Section 6.2.

The *output_graph_file* filename refers to a distributed graph, following the same semantics.

Options

`-c`*num*
: Set the minimum number of common neighbors that defines the adjacency relationship between the elements in the element graph. Two elements in the element graph will be connected if the number of nodes they have in common in the input auxiliary mesh graph is greater than or equal to `num`. This value is set to 1 by default.

`-h`
: Display the program synopsis.

`-V`
: Print the program version and copyright.

6.4.2 `dgmap` / `dgpart`

Synopsis

`dgmap` [*input_graph_file* [*input_target_file* [*output_mapping_file* [*output_log_file*]]]] *options*

`dgpart` *number_of_parts* [*input_graph_file* [*output_mapping_file* [*output_log_file*]]] *options*

Description

The `dgmap` program is the parallel static mapper. It uses a static mapping strategy to compute a mapping of the given source graph to the given target architecture. The implemented algorithms aim at assigning source graph vertices to target vertices such that every target vertex receives a set of source vertices of summed weight proportional to the relative weight of the target vertex in the target architecture, and such that the communication cost function f_C is minimized (see Section 3.1.2 for the definition and rationale of this cost function).

Since its main purpose is to provide mappings that exhibit high concurrency for communication minimization in the mapped application, it comprises a parallel implementation of the dual recursive bipartitioning algorithm [33], as

well as all of the sequential static mapping methods used by its sequential counterpart `gmap`, to be used on subgraphs located on single processors.

`dgpart` is a simplified interface to `dgmap`, which performs graph partitioning instead of static mapping. Consequently, the desired number of parts has to be provided, in lieu of the target architecture.

The `-b` and `-c` options allow the user to set preferences on the behavior of the mapping strategy which is used by default. The `-m` option allows the user to define a custom mapping strategy.

The *input_graph_file* filename can refer either to a centralized or to a distributed graph, according to the semantics defined in Section 6.2. The mapping 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.

`-b`*rat*
: Set the maximum load imbalance ratio to *rat*, which should be a value comprised between 0 and 1. This option can be used in conjunction with option `-c`, but is incompatible with option `-m`.

`-c`*flags*
: Tune the default mapping strategy according to the given preference flags. Some of these flags are antagonistic, while others can be combined. See Section 7.4.1 for more information. The currently available flags are the following.

  `b` Enforce load balance as much as possible.

  `q` Privilege quality over speed. This is the default behavior.

  `s` Privilege speed over quality.

  `t` Use only safe methods in the strategy.

  `x` Favor scalability.

  This option can be used in conjunction with option `-b`, but is incompatible with option `-m`. The resulting strategy string can be displayed by means of the `-vs` option.

`-h`
: Display the program synopsis.

`-m`*strat*
: Apply parallel static mapping strategy *strat*. The format of parallel mapping strategies is defined in section 7.4.2. This option is incompatible with options `-b` and `-c`.

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

`-s`*obj*
: Mask source edge and vertex weights. This option allows the user to "unweight" weighted source graphs by removing weights from edges and vertices at loading time. *obj* may contain several of the following switches.

  `e` Remove edge weights, if any.

  `v` Remove vertex weights, if any.

`-V` Print the program version and copyright.

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

- `a` Memory allocation information.
- `m` Mapping information, similar to the one displayed by the `gmtst` program of the sequential SCOTCH distribution.
- `s` Strategy information. This parameter displays the default mapping strategy used by `gmap`.
- `t` Timing information.

6.4.3 `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 [14], but sequential classical [31] and state-of-the-art [39] 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 `-c` option allows the user to set preferences on the behavior of the ordering strategy which is used by default.

The *input_graph_file* filename can refer either to a centralized or to a distributed graph, according to the semantics defined in Section 6.2. 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.

`-c`*flags*
Tune the default ordering strategy according to the given preference flags. Some of these flags are antagonistic, while others can be combined. See Section 7.4.1 for more information. The resulting strategy string can be displayed by means of the `-vs` option.

- `b` Enforce load balance as much as possible.

`q` Privilege quality over speed. This is the default behavior.

`s` Privilege speed over quality.

`t` Use only safe methods in the strategy.

`x` Favor scalability.

`-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* [35].

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 7.4.4.

`-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.

`a` Memory allocation information.

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

`t` Timing information.

6.4.4 `dgpart`

Synopsis

`dgpart` [*number_of_parts* [*input_graph_file* [*output_mapping_file* [*output_log_file*]]]] *options*

Description

The `dgpart` program is the parallel graph partitioner. It is in fact a shortcut for the `dgmap` program, where the number of parts is turned into a complete graph with same number of vertices which is passed to the static mapping routine.

Save for the *number_of_parts* parameter which replaces the *input_target_file*, the parameters of `dgpart` are identical to the ones of `dgmap`. Please refer to its manual page, in Section 6.4.2, for a description of all of the available options.

6.4.5 `dgscat` / `gscat`

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 6.2.

`dgscat` has a sequential counterpart, called `gscat`. The latter operates by processing the source graph file on the fly, and does not perform any consistency checking on the output it produces.

Options

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

`-h` Display the program synopsis.

`-i`*nbr*
: For gscat only. Create an imbalanced distributed graph, in which the distributed graph files for all processes will receive *nbr* vertices of the graph, save for the file for the last process, which will receive all the remaining vertices.

`-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.

6.4.6 `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.

7 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 `libptscotch.a` and `libptscotcherr.a`. All of the existing parallel routines belong to five distinct classes:

- distributed source graph handling routines, which serve to declare, build, load, save, and check the consistency of distributed source graphs;
- distributed source mesh handling routines, which serve to declare, build, and loaddistributed source meshes;
- strategy handling routines, which allow the user to declare and build parallel mapping and ordering strategies;
- parallel graph partitioning and static mapping routines, which allow the user to declare, compute, and save distributed static mappings of distributed source graphs;
- 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* [35]. Their use is recalled in Section 7.16.

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 7.20.

7.1 Using multi-threading

Scotch and PT-Scotch comprise several multi-threaded algorithms, which can significally reduce overall computation time. Also, thread management in Scotch is now dynamic, and is controlled by `SCOTCH_Context` objects. These objects define user-configurable execution contexts, in which libPTScotch library routines can be executed independently from others.

7.1.1 Compiling at proper thread level

Two levels of multi-threading are available: multi-threading with a single thread handling MPI communications, and multi-threading with several threads being able to perform MPI communication simultaneously. This behavior is controlled at compile-time by flags "`-DSCOTCH_PTHREAD`" and "`-DSCOTCH_PTHREAD_MPI`". The selection of the proper compilation flags depends on the capabilities of the platform MPI implementation. Alternately, the compilation flag "`-DSCOTCH_MPI_ASYNC_COLL`" can be used to replace threaded synchronous communication primitives by non-threaded asynchronous communication primitives.

7.1.2 Running at proper thread level

Since PT-Scotch is based on the MPI API, all processes must call some flavor of `MPI_Init` before using any routine of the library that performs communication.

If PT-Scotch has been compiled without the "`-DSCOTCH_PTHREAD`" flag, a call to the simple `MPI_Init` routine will suffice. Else, the extended `MPI_Init_thread` initialization routine has to be used. If PT-Scotch has been compiled with the "`-DSCOTCH_PTHREAD_MPI`" flag as well, and the MPI library is launched at the `MPI_THREAD_MULTIPLE` level, PT-Scotch will take advantage of this to run threaded versions of some of its algorithms across the MPI nodes.

Thread binding is essential to achieve good performance of multi-threaded programs. In Scotch, the only thread binding mechanism implemented to date relies on the Linux binding API. Please make sure to compile with flag "`-DCOMMON_PTHREAD_AFFINITY_LINUX`" set whenever possible, to benefit from these features.

7.1.3 Launching with threads

Running a multi-threaded version of PT-Scotch on top of MPI requires great care, not to degrade performance. The key issue is to make sure that, at run time, MPI processes that run on the same computer are bound to disjoint subsets of cores.

Indeed, when several MPI processes run on the same computer without implementing exclusive binding, every `SCOTCH_Context` object associated with an MPI process will consider it can use all of the computer cores. Consequently, as many threads will be created and bound to each of the cores as there are MPI processes on the computer, resulting in extremely poor performance.

On many-core platforms, it is up to the user to decide how many MPI processes, and how many threads per MPI process, are to be used. This is usually controlled by options of the `mpirun` command. For instance, in the OpenMPI environment, one may use options such as "`--map-by socket`", "`--bind-to socket`", "`--bind-to core`", etc. The result of the bindings can be displayed using the "`--report-bindings`" option. For instance, in this environment, one can use commands such as:

```
% mpirun -np 2 --map-by socket --bind-to socket <program>
% mpirun -np 4 --map-by socket:PE=7 --bind-to core <program>
```

% mpirun -np 8 --map-by socket:PE=3 --bind-to core <program>
to achieve the desired MPI process and thread layout.

When SCOTCH is compiled with flag "-DCOMMON_PTHREAD_AFFINITY_LINUX" set, each of the SCOTCH_Context objects that are attached to an MPI process running a libPTSCOTCH function will create its servant threads only on the cores that are bound to the main thread of this MPI process. Hence, strict core binding exclusion will be enforced across MPI processes and threads, so that each core will host at most only one thread.

7.2 Calling the routines of libSCOTCH

7.2.1 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 7.5, can be called from C by using the following code.

```
#include <stdio.h>
#include <mpi.h>
#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 */
  }
  ...
```

Since "ptscotch.h" uses several system and communication objects which are declared in "stdio.h" and "mpi.h", respectively, these latter files must be included beforehand in your application code.

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.

7.2.2 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 PXFFILENO in the normalized POSIX Fortran API, and files which use it should include the USE IFPOSIX directive whenever necessary. An alternate, non normalized, function also exists in most Unix implementations of Fortran, and is called FNUM.

For instance, the SCOTCH_dgraphInit and SCOTCH_dgraphLoad routines, described in sections 7.5.14 and 7.6.1, 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), PXFFILENO (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.

7.2.3 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 7.16 for more information on error handling.

Programs that use both sequential and parallel routines of Scotch need only be linked against the parallel version of the library, as it also contains an adapted version of the sequential routines. The reason why the sequential routines are duplicated in the parallel PT-Scotch library is because they are slightly modified so as to keep track of the parallel environment. This allows one to create "multi-sequential" routines that can exchange data with other processes, for instance. Because the libScotch data structures contain extra parameters, never mix the `scotch.h` sequential include file and the `libptscotch.a` parallel library, as the latter expects Scotch data structures to be larger than the ones defined in the sequential include file. Consequently, when using only sequential routines in a sequential program, include the `scotch.h` file only and link the program against the sequential `libscotch.a` library only. When using only parallel routines, or when using a mix of sequential and parallel routines, include the `ptscotch.h` file only and link the program against the parallel `libptscotch.a` library only. When using only sequential routines in a parallel program, both options can be used.

7.2.4 Machine word size issues

Graph indices are represented in Scotch as integer values of type `SCOTCH_Num`. By default, this type equates to the `int` C type, that is, an integer type of size equal to that of the machine word. However, it can represent any other integer type. Indeed, the size of the `SCOTCH_Num` integer type can be coerced to 32 or 64 bits by using the "`-DINTSIZE32`" or "`-DINTSIZE64`" compilation flags, respectively, or else by using the "`-DINT=`" definition (see Section 8.3 for more information on the setting of these compilation flags).

This may, however, pose a problem with MPI, the interface of which is based on the regular `int` type. PT-Scotch has been coded so as to avoid typecast bugs, but overflow errors may result from the conversion of values of a larger integer type into `int`s when handling communication buffer indices.

Consequently, the C interface of Scotch uses two types of integers. Graph-related quantities are passed as `SCOTCH_Num`s, while system-related values such as

file handles, as well as return values of libScotch routines, are always passed as `int`s.

Because of the variability of library integer type sizes, one must be careful when using the Fortran interface of Scotch, as it does not provide any prototyping information, and consequently cannot produce any warning at link time. In the manual pages of the libScotch routines, Fortran prototypes are written using three types of `INTEGER`s. As for the C interface, the regular `INTEGER` type is used for system-based values, such as file handles and MPI communicators, as well as for return values of the libScotch routines, while the `INTEGER*`*num* type should be used for all graph-related values, in accordance to the size of the `SCOTCH_Num` type, as set by the "`-DINTSIZE`*x*" compilation flags. Also, the `INTEGER*`*idx* type represents an integer type of a size equivalent to that of a `SCOTCH_Idx`, as set by the "`-DIDXSIZE`*x*" compilation flags. Values of this type are used in the Fortran interface to represent arbitrary array indices which can span across the whole address space, and consequently deserve special treatment.

In practice, when Scotch is compiled on a 32-bit architecture so as to use 64-bit `SCOTCH_Num`s, graph indices should be declared as `INTEGER*8`, while error return values should still be declared as plain `INTEGER` (that is, `INTEGER*4`) values. On a 32_64-bit architecture, irrespective of whether `SCOTCH_Num`s are defined as `INTEGER*4` or `INTEGER*8` quantities, the `SCOTCH_Idx` type should always be defined as a 64-bit quantity, that is, an `INTEGER*8`, because it stores differences between memory addresses, which are represented by 64-bit values. The above is no longer a problem if Scotch is compiled such that `int`s equate 64-bit integers. In this case, there is no need to use any type coercing definition.

The MeTiS v3 compatibility library provided by Scotch can also run on a 64-bit architecture. Yet, if you are willing to use it this way, you will have to replace all `int`'s that are passed to the MeTiS routines by 64-bit integer `SCOTCH_Num` values (even the option configuration values). However, in this case, you will no longer be able to link against the service routines of the genuine MeTiS/ParMeTiS v3 library, as they are only available as a 32-bit implementation.

7.3 Data types

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.

7.3.1 `SCOTCH_Dgraph` distributed graph type

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 9 and 10. 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 `procvrttab`$[p+1] \geq$ (`procvrttab`$[p]$ + `proccnttab`$[p]$), 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, `procvrttab`$[p+1] =$ (`procvrttab`$[p]$ + `proccnttab`$[p]$).

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 `proccnttab`$[p]$.

`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 `baseval` $\leq i <$ (`baseval` + `vertlocnbr`), `vendloctab`[i] − `vertloctab`[i] 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 9, `vertloctab` is of size `vertlocnbr` + 1 and `vendloctab` points to `vertloctab` + 1. For these graphs , called "compact edge array graphs", or "compact graphs" for short, `vertloctab` is sorted in ascending order, `vertloctab[baseval]` = `baseval` and `vertloctab[baseval` + `vertlocnbr]` = (`baseval` + `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 ($\max_i$(`vendloctab`[i]) − `baseval`), holding the adjacency array of every local vertex. For any local vertex i, with `baseval` $\leq i <$ (`baseval` + `vertlocnbr`), the global indices of the neighbors of i are stored in `edgeloctab` from `edgeloctab[vertloctab[`i`]]` to `edgeloctab[vendloctab[`i`]` − 1`]`, 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_dgraph Halo` routine documented in Section 7.7.2.

`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.

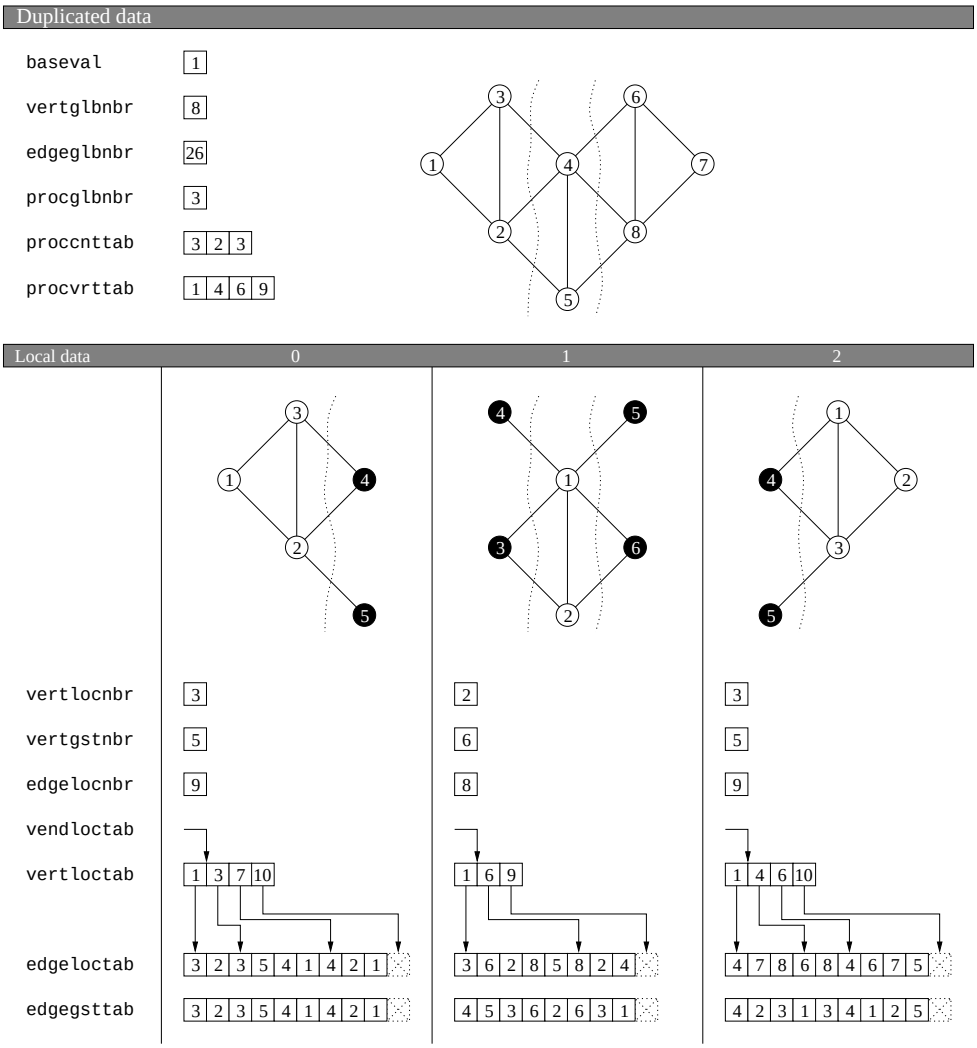


Figure 9: Sample distributed graph and its description by LIBSCOTCH arrays using a continuous numbering and compact edge arrays. Numbers within vertices are vertex indices. Top graph is a global view of the distributed graph, labeled with global, continuous, indices. Bottom graphs are local views labeled 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. `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 `veloloctab` and `edloloctab` are not represented.

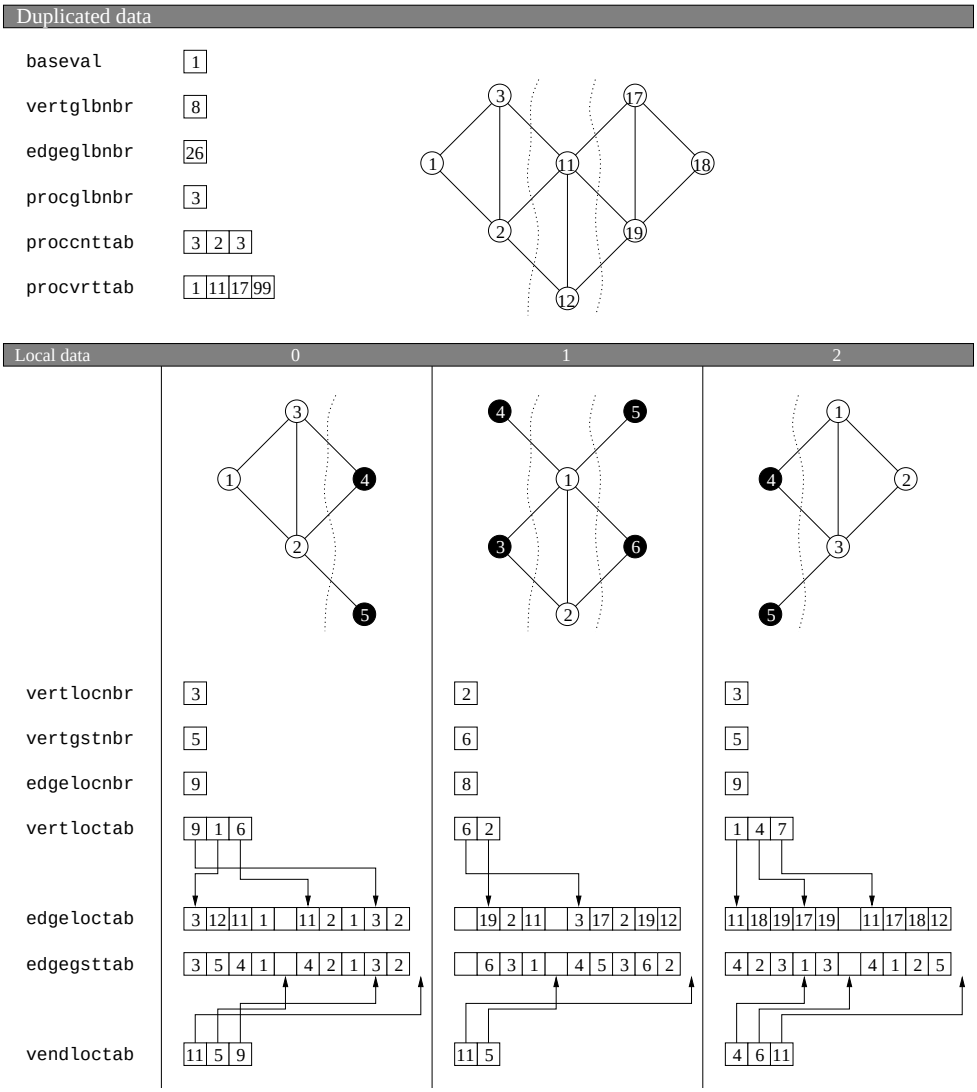


Figure 10: Adjacency structure of the sample graph of Figure 9 with a disjoint edge array and a discontinuous ordering. Both `vertloctab` and `vendloctab` are of size `vertlocnbr`. This allows for the handling of dynamic graphs, the structure of which can evolve with time.

Dynamic graphs can be handled elegantly by using the `vendloctab` and `proc vrttab` 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.

7.3.2 `SCOTCH_Dmesh` distributed mesh type

The distributed mesh data type is currently a prototype, which has no official support. Its main use is to host data that will eventually serve to compute a `SCOTCH_Dgraph` distributed dual graph. Unlike other Scotch and PT-Scotch data structures, such as the `SCOTCH_Mesh` centralized data structure, which are symmetric, this mesh data type only provides adjacency lists for elements: the list of nodes connected to a given element is available, while the node-to-element adjacency is currently not. Hence, at the time being, there is no possibility to provide node weights, since node vertices have no existence. Element weights are not handled to date either, but that may change in the future if the need arises. Due to the prototype nature of this data type, its internal structure may be subject to changes at any time.

As said, the `SCOTCH_Dmesh` structure only holds the minimum set of information needed to perform dual graph computation. 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 element vertex data is in $O(v_e + p)$, where v_e is the overall number of vertices in the distributed mesh and p the number of processes, and the sum of all of the edge data is in $O(e_e)$, where e_e is the overall number of element-to-node arcs in the distributed mesh. No halo information is present.

Distributed source meshes are described by means of adjacency lists. The description of a distributed mesh requires several SCOTCH_Num scalars and arrays, as illustrated in Figure 11. Some of these data are said to be global, and are duplicated on every process that holds part of the distributed mesh; their names contain the "glb" infix. Others are local, that is, their value may differ for each process; their names contain the "loc" infix. Global data have the following meaning:

baseval
: Base value for all array indexings.

velmglbnbr
: Overall number of element vertices in the distributed mesh.

eelmglbnbr
: Overall number of element-to-node arcs in the distributed mesh, which is also the overall number of node-to-element arcs.

vnodglbnbr
: Overall number of node vertices in the distributed mesh.

procglbnbr
: Overall number of processes that share distributed mesh data.

prelvrttab
: Array holding the global indices from which the element vertices of every process are numbered. For optimization purposes, this array has an extra slot, to record the after-last based number of elements. Since, to date, distributed meshes are compact data structures, `prelvrttab[0]` = `baseval`, `prelvrttab[procglbnbr]` = (`velmglbnbr` + `baseval`), and, for each process index p, and with respect to its local variable `velmlocnbr`, `prelvrttab`[$p+1$] = (`prelvrttab`[p] + `velmlocnbr`) holds.

Local data have the following meaning:

velmlocnbr
: Number of local element vertices borne by the given process.

eelmlocnbr
: Number of local element-to-node arcs borne by the given process.

velmloctab
: Array of start indices in `eelmloctab` of vertex adjacency sub-arrays. Since distributed mesh numbering is compact, this array possesses and extra slot, to record the after-last based number of local element-to-node arcs, that is, `velmloctab[velmlocnbr]` = (`eelmlocnbr[` + `baseval`) holds.

eelmloctab
: Array, of a size equal to `eelmlocnbr`, holding the adjacency array of every local element vertex, in the form of a set of based, global node vertex indices. For any local element vertex i, with `baseval` $\leq i <$ (`baseval` + `velmlocnbr`), the global indices of the node neighbors of i are stored in `eelmloctab` from `eelmloctab[velmloctab[`i`]]` to `eelmloctab[velmloctab[`$i+1$`] - 1]`, inclusive.

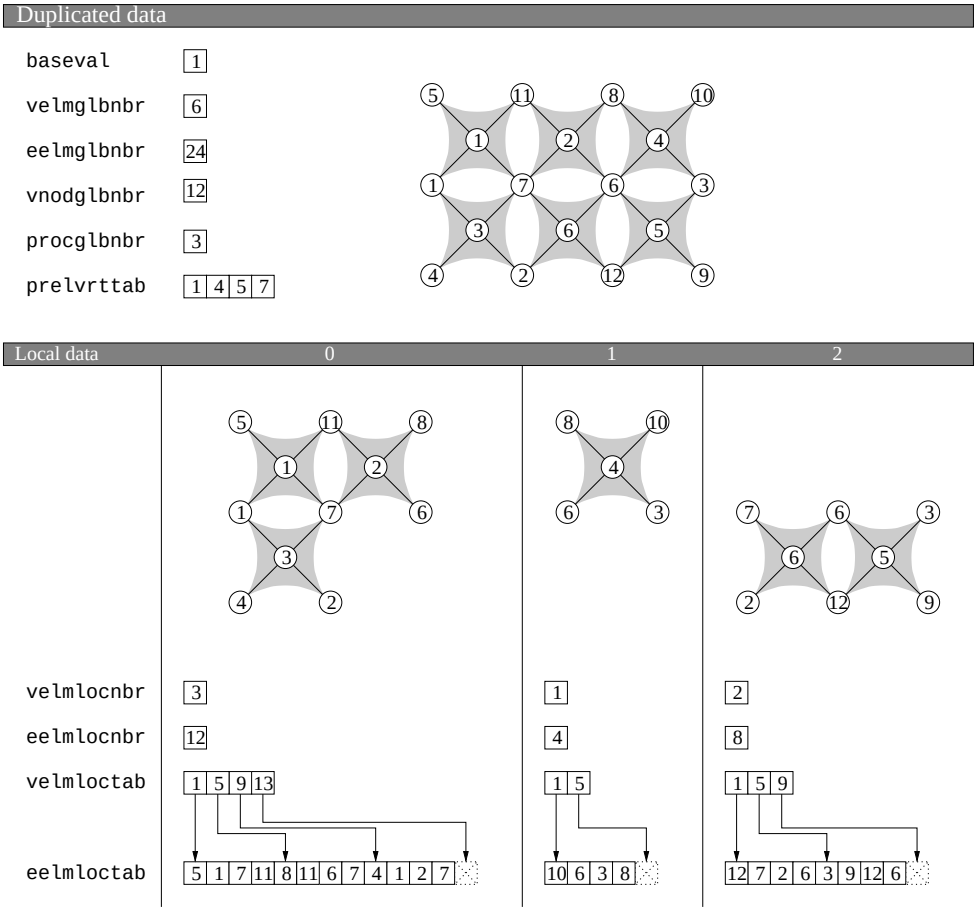


Figure 11: Sample distributed mesh and its description by LIBSCOTCH arrays. Numbers within vertices are element and node vertex indices. Top mesh is a global view of the distributed mesh, labeled with global, continuous, indices. Bottom graphs are local views. Since the element edge array is compact, all `velmloctab` arrays are of size `velmlocnbr` + 1. `eelmloctab` edge arrays hold global indices of end node vertices. `eelmlocnbr` is the local number of element-to-node arcs.

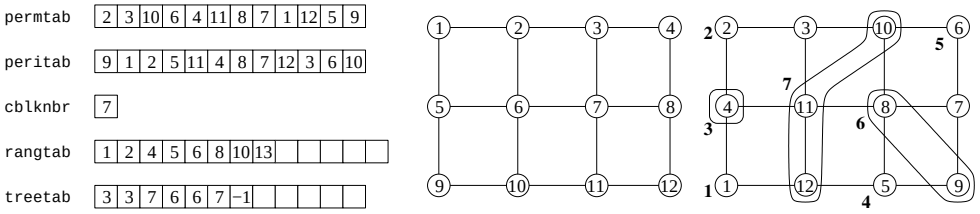


Figure 12: Arrays resulting from the ordering by complete nested dissection of a 4 by 3 grid based from 1. Leftmost grid is the original grid, and righmost grid is the reordered grid, with separators shown and column block indices written in bold.

7.3.3 `SCOTCH_Dordering` distributed block ordering type

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 12 for a complete example.

7.4 Strategy strings

The behavior of the static mapping and 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

7.4.1 Using default strategy strings

While strategy strings can be built by hand, according to the syntax given in the next sections, users who do not have specific needs can take advantage of default strategies already implemented in the libScotch, which will yield very good results in most cases. By doing so, they will spare themselves the hassle of updating their strategies to comply to subsequent syntactic changes, and they will benefit from the availability of new partitioning or ordering methods as soon as they are made available.

The simplest way to use default strategy strings is to avoid specifying any. By initializing a strategy object, by means of the `SCOTCH_stratInit` routine, and by using the initialized strategy object as is, without further parametrization, this object will be filled with a default strategy when passing it as a parameter to the next partitioning or ordering routine to be called. On return, the strategy object will contain a fully specified strategy, tailored for the type of operation which has been requested. Consequently, a fresh strategy object that was used to partition a graph cannot be used afterward as a default strategy for calling an ordering routine, for instance, as partitioning and ordering strategies are incompatible.

The libScotch also provides helper routines which allow users to express their preferences on the kind of strategy that they need. These helper routines, which are of the form `SCOTCH_strat*Build`, tune default strategy strings according to parameters provided by the user, such as the requested number of parts (used as a hint to select the most efficient partitioning routines), the desired maximum load imbalance ratio, and a set of preference flags. While some of these flags are antagonistic, most of them can be combined, by means of addition or "binary or" operators. These flags are the following. They are grouped by application class.

Global flags

`SCOTCH_STRATDEFAULT`
: Default behavior. No flags are set.

`SCOTCH_STRATBALANCE`
: Enforce load balance as much as possible.

`SCOTCH_STRATQUALITY`
: Privilege quality over speed.

`SCOTCH_STRATSAFETY`
: Do not use methods that can lead to the occurrence of problematic events, such as floating point exceptions, which could not be properly handled by the calling software.

`SCOTCH_STRATSPEED`
: Privilege speed over quality.

Ordering flags

`SCOTCH_STRATLEVELMAX`
: Create at most the prescribed levels of nested dissection separators. If the number of levels is less than the logarithm of the number of processing elements used, distributed pieces of the separated subgraph may have to be centralized so that the leaves can be ordered, which may result in memory shortage.

`SCOTCH_STRATLEVELMIN`
: Create at least the prescribed levels of nested dissection separators. When used in conjunction with `SCOTCH_STRATLEVELMAX`, the exact number of nested dissection levels will be performed, unless the graph to order is too small.

`SCOTCH_STRATLEAFSIMPLE`
: Order nested dissection leaves as cheaply as possible.

`SCOTCH_STRATSEPASIMPLE`
: Order nested dissection separators as cheaply as possible.

7.4.2 Parallel mapping strategy strings

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

`(`*strat*`)`
: Grouping operator. The strategy enclosed within the parentheses is treated as a single mapping 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 mapping task, 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 recursion 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 mapping method. Available parallel mapping methods are listed below.

The currently available parallel mapping methods are the following.

`r`
: Dual recursive bipartitioning method. The parameters of the dual recursive bipartitioning method are given below.

  `seq=`*strat*
  : Set the sequential mapping strategy that is used on every centralized subgraph of the recursion tree, once the dual recursive bipartitioning process has gone far enough such that the number of processes handling some subgraph is restricted to one.

  `sep=`*strat*
  : Set the parallel graph bipartitioning strategy that is used on every current job of the recursion tree. Parallel graph bipartitioning strategies are described below, in section 7.4.3.

7.4.3 Parallel graph bipartitioning strategy strings

A parallel graph bipartitioning strategy is made of one or several parallel graph bipartitioning 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 bipartition of the two that are obtained by the distinct application of *strat1* and *strat2* to the current bipartition.

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

(*strat*)

Grouping operator. The strategy enclosed within the parentheses is treated as a single bipartitioning 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 active graph, 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 bipartition or multi-level 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.

`load0`

The vertex load of the first subset of the current bipartition of the current graph. 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 frontier, runs a

parallel graph bipartitioning strategy on this graph, and prolongs back the new bipartition to the current graph. This method is primarily used to run bipartition refinement methods during the uncoarsening phase of the multi-level parallel graph bipartitioning method. The parameters of the band method are listed below.

`bnd=`*strat*
: Set the parallel graph bipartitioning strategy to be applied to the band graph.

`org=`*strat*
: Set the parallel graph bipartitioning strategy to be applied to the full distributed graph if the band graph could not be extracted.

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

`d`
: Parallel diffusion method. This method, presented in its sequential formulation in [34], flows two kinds of antagonistic liquids, scotch and anti-scotch, from two source vertices, and sets the new frontier as the limit between vertices which contain scotch and the ones which contain anti-scotch. Because selecting the source vertices is essential to the obtainment of useful results, this method has been hard-coded so that the two source vertices are the two vertices of highest indices, since in the band method these are the anchor vertices which represent all of the removed vertices of each part. Therefore, this method must be used on band graphs only, or on specifically crafted graphs. Applying it to any other graphs is very likely to lead to extremely poor results. The parameters of the diffusion bipartitioning method are listed below.

  `dif=`*rat*
  : Fraction of liquid which is diffused to neighbor vertices at each pass. To achieve convergence, the sum of the `dif` and `rem` parameters must be equal to 1, but in order to speed-up the diffusion process, other combinations of higher sum can be tried. In this case, the number of passes must be kept low, to avoid numerical overflows which would make the results useless.

  `pass=`*nbr*
  : Set the number of diffusion sweeps performed by the algorithm. This number depends on the width of the band graph to which the diffusion method is applied. Useful values range from 30 to 500 according to chosen `dif` and `rem` coefficients.

  `rem=`*rat*
  : Fraction of liquid which remains on vertices at each pass. See above.

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

  `asc=`*strat*
  : Set the strategy that is used to refine the distributed bipartition obtained at ascending levels of the uncoarsening phase by prolongation of the bipartition 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 bipartition 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 graph bipartitioning method is applied independently to each of them. Then, the best bipartition found is prolonged 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 edge separation strategy that is used to refine the bipartition of the centralized graph. For a description of all of the available sequential bipartitioning methods, please refer to the Scotch *User's Guide* [35].

`x` Load balance enforcement method. This method moves as many vertices from the heaviest part to the lightest one so as to reduce load imbalance as much as possible, without impacting communication load too negatively. The only parameter of this method is listed below.

`sbbt=`*nbr*
Number of sub-buckets to sort communication gains. 5 is a common value.

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

7.4.4 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 7.4.5.

`q` Sequential ordering method. The distributed graph is gathered onto a single process which runs a sequential ordering strategy. The only parameter of the sequential method is given below.

`strat=`*strat*
: Set the sequential ordering strategy that is applied to the centralized graph. For a description of all of the available sequential ordering methods, please refer to the SCOTCH *User's Guide* [35].

`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

7.4.5 Parallel node separation strategy strings

A parallel 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 prolongs 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 prolongation 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 prolonged 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* [35].

`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.

7.5 Distributed graph handling routines

7.5.1 SCOTCH_dgraphAlloc

Synopsis

```
SCOTCH_Dgraph * SCOTCH_dgraphAlloc (void)
```

Description

The `SCOTCH_dgraphAlloc` function allocates a memory area of a size sufficient to store a `SCOTCH_Dgraph` structure. It is the user's responsibility to free this memory when it is no longer needed. The allocated space must be initialized before use, by means of the `SCOTCH_dgraphInit` routine.

Return values

`SCOTCH_dgraphAlloc` returns the pointer to the memory area if it has been successfully allocated, and `NULL` else.

7.5.2 SCOTCH_dgraphBand

Synopsis

```
int SCOTCH_dgraphBand (SCOTCH_Dgraph * const     orggrafptr,
                       const SCOTCH_Num          fronlocnbr,
                       const SCOTCH_Num * const  fronloctab,
                       const SCOTCH_Num          distval,
                       SCOTCH_Dgraph * const     bndgrafptr)

scotchfdgraphband (doubleprecision (*)  orggrafdat,
                   integer*num          seedlocnbr,
                   integer*num (*)      seedloctab,
                   integer*num          distval,
                   doubleprecision (*)  bndgrafdat,
                   integer              ierr)
```

Description

The `SCOTCH_dgraphBand` routine creates in the `SCOTCH_Dgraph` structure pointed to by `bndgrafptr` a distributed band graph induced from the `SCOTCH_Dgraph` pointed to by `orggrafptr`. The distributed band graph will contain all the vertices of the original graph located at a distance smaller than or

equal to `distval` from any vertex provided in the `seedloctab` lists of seed vertices.

On each process, the `seedloctab` array should contain the local indices of the local vertices that will serve as seeds. The number of such local vertices is passed to `SCOTCH_dgraphBand` in the `seedlocnbr` value. The size of the `seedloctab` array should be at least equal to the number of local vertices of the original graph, as it is internally used as a queue array. Hence, no user data should be placed immediately after the `seedlocnbr` values in the array, as they are most likely to be overwritten by `SCOTCH_dgraphBand`.

`bndgrafptr` must have been initialized with the `SCOTCH_dgraphInit` routine before `SCOTCH_dgraphBand` is called. The communicator that is passed to it can either be the communicator used by the original graph `orggrafptr`, or any congruent communicator created by using `MPI_Comm_dup` on this communicator. Using a distinct communicator for the induced band graph allows subsequent library routines to be called in parallel on the two graphs after the band graph is created.

Induced band graphs have vertex labels attached to each of their vertices, in the `vlblloctab` array. If the original graph had vertex labels attached to it, band graph vertex labels are the labels of the corresponding vertices in the original graph. Else, band graph vertex labels are the global indices of corresponding vertices in the original graph.

Depending on original graph vertex and seed distributions, the distribution of induced band graph vertices may be highly imbalanced. In order for further computations on this distributed graph to scale well, a redistribution of its data may be necessary, using the `SCOTCH_dgraphRedist` routine.

Return values

`SCOTCH_dgraphBand` returns 0 if the band graph structure has been successfully created, and 1 else.

7.5.3 `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*num          baseval,
                    integer*num          vertlocnbr,
                    integer*num          vertlocmax,
                    integer*num (*)      vertloctab,
                    integer*num (*)      vendloctab,
                    integer*num (*)      veloloctab,
                    integer*num (*)      vlblloctab,
                    integer*num          edgelocnbr,
                    integer*num          edgelocsiz,
                    integer*num (*)      edgeloctab,
                    integer*num (*)      edgegsttab,
                    integer*num (*)      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.

Note that, for PT-Scotch to operate properly, either all the arrays of a kind must be set to null on all processes, or else all of them must be non null. This is mandatory because some algorithms require that collective communication operations be performed when some kind of data is present. If some processes

considered that the arrays are present, and start such communications, while others did not, a deadlock would occur. In most cases, this situation will be anticipated and an error message will be issued, stating that graph data are inconsistent.

The situation above may accidentally arise when some processes don't own any edge or vertex. In that case, depending on the implementation, a user `malloc` of size zero may return a null pointer rather than a non null pointer to an area of size zero, leading to the aforementioned inconsistencies. In order to avoid this problem, it is necessary to ensure that no null pointer will be returned, even in the case when zero bytes are requested. A workaround can be to call `malloc (x | 4)` instead of `malloc (x)`. The "`| 4`" will consume only 4 extra bytes at most, depending on the value of `x`.

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.

7.5.4 SCOTCH_dgraphBuildGrid3D

Synopsis

```
int SCOTCH_dgraphBuildGrid3D (SCOTCH_Dgraph *   grafptr,
                              const SCOTCH_Num  baseval,
                              const SCOTCH_Num  dimxval,
                              const SCOTCH_Num  dimyval,
                              const SCOTCH_Num  dimzval,
                              const SCOTCH_Num  incrval,
                              const SCOTCH_Num  flagval)
```

```
scotchfdgraphbuildgrid3d (doubleprecision (*)  grafdat,
                          integer*num          baseval,
                          integer*num          dimxval,
                          integer*num          dimyval,
                          integer*num          dimzval,
                          integer*num          incrval,
                          integer*num          flagval,
                          integer              ierr)
```

Description

The SCOTCH_dgraphBuildGrid3D routine fills the graph structure pointed to by grafptr with the description of a 3D grid of dimensions dimxval, dimyval, and dimzval, using baseval as the base value for vertex and edge indices. It is mostly used to build graphs of arbitrary size for testing other PT-Scotch routines.

The incrval value is the increment value for vertex indices. It should be a co-prime with dimxval, dimyval, and dimzval, so that all vertex indices are enumerated before looping back to the initial vertex index. When equal to 1, all vertex indices are enumerated consecutively, and vertex labels are not used. When greater than 1, vertex indices are enumerated using this stride value, and vertex labels are used.

The flagval value allows the user to combine several flags to indicate the type of grid that should be produced:

SCOTCH_DGRAPHBUILDGRID3DGRID
: Create a plain grid graph (*i.e.*, not a torus).

SCOTCH_DGRAPHBUILDGRID3DTORUS
: Create a torus graph, in which vertices on the boundaries of the grid are connected to the vertices of the opposite side.

SCOTCH_DGRAPHBUILDGRID3DNGB6
: Create a 6-neighbor mesh, without diagonal edges.

SCOTCH_DGRAPHBUILDGRID3DNGB26
: Create a 26-neighbor mesh, with diagonal edges in all directions.

SCOTCH_DGRAPHBUILDGRID3DVERTLOAD
: Place arbitrary loads on graph vertices.

SCOTCH_DGRAPHBUILDGRID3DEDGELOAD
: Place arbitrary loads on graph edges.

Return values

SCOTCH_dgraphBuildGrid3D returns 0 if the graph structure has been successfully created, and 1 else.

7.5.5 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.

7.5.6 SCOTCH_dgraphCoarsen

Synopsis

```
int SCOTCH_dgraphCoarsen (SCOTCH_Dgraph * const  finegrafptr,
                          const SCOTCH_Num       coarnbr,
                          const double           coarrat,
                          const SCOTCH_Num       flagval,
                          SCOTCH_Dgraph * const  coargrafptr,
                          SCOTCH_Num * const     multloctab)

scotchfdgraphcoarsen (doubleprecision (*)  finegrafdat,
                      integer*num          coarnbr,
                      doubleprecision      coarrat,
                      integer*num          flagval,
                      doubleprecision (*)  coargrafdat,
                      integer*num (*)      multloctab,
                      integer              ierr)
```

Description

The SCOTCH_dgraphCoarsen routine creates in the SCOTCH_Dgraph structure pointed to by coargrafptr a distributed coarsened graph from the SCOTCH_Dgraph pointed to by finegrafptr. The coarsened graph is created only if it is comprises more than coarnbr vertices, or if the coarsening ratio is lower than coarrat. Valid coarsening ratio values range from 0.5 (in the case of a perfect matching) to 1.0 (if no vertex could be coarsened). Classical threshold values range from 0.7 to 0.8.

The flagval flag specifies the type of coarsening. Several groups of flags can be combined, by means of addition or "binary or" operators. When SCOTCH_COARSENNOMERGE is set, isolated vertices are never merged with other vertices. This preserves the topology of the graph, at the expense of a higher coarsening ratio. When SCOTCH_COARSENFOLD or SCOTCH_COARSENFOLDDUP are set, if a coarsened graph is created, it is folded onto half of the processes of the initial communicator. In the case of SCOTCH_COARSENFOLDDUP, a second

copy is created (duplicated) onto the other half. The two copies may not be identical, if the number of processors of the finer graph is odd.

The `multloctab` multinode array will contain pairs of consecutive `SCOTCH_Num` values, representing the global indices of the two fine vertices that have been coarsened into each of the local coarse vertices. When a fine vertex has not been matched, the two indices in the multinode array for the corresponding coarse vertex are equal to the index of the single fine vertex. The `multloctab` array must be large enough to store node matching local data for the resulting coarsened (and possibly folded) graph. The `SCOTCH_dgraphCoarsenVertLocMax` routine (see Section 7.5.7) provides an upper bound on the number of coarse (and possibly folded) vertices located on the local process, which has to be doubled to obtain an upper bound on the number of `SCOTCH_Num`s contained in `multloctab`. Note that, in case of folding, a redistribution of vertices is performed, so that the number of local coarse vertices may be greater that the number of local fine vertices.

`coargrafptr` must have been initialized with the `SCOTCH_dgraphInit` routine before `SCOTCH_dgraphCoarsen` is called. The communicator that is passed to it can either be the communicator used by the fine graph `finegrafptr`, or any congruent communicator created by using `MPI_Comm_dup` on this communicator. Using a distinct communicator for the coarsened subgraph allows subsequent library routines to be called in parallel on the two graphs after the coarse graph is created.

Depending on the way vertex mating is performed, the distribution of coarsened graph vertices may be imbalanced. In order for further computations on this distributed graph to scale well, a redistribution of its data might be necessary, using the `SCOTCH_dgraphRedist` routine.

Return values

`SCOTCH_dgraphCoarsen` returns 0 if the coarse graph structure has been successfully created, 1 if the coarse graph was not created because it did not enforce the threshold parameters, and 2 on error.

7.5.7 `SCOTCH_dgraphCoarsenVertLocMax`

Synopsis

```
SCOTCH_Num SCOTCH_dgraphCoarsenVertLocMax (SCOTCH_Dgraph * const  finegrafptr,
                                           const SCOTCH_Num       flagval)

scotchfdgraphcoarsenvertlocmax (doubleprecision (*)  finegrafdat,
                                integer*num          flagval,
                                integer*num          coarvertlocmax)
```

Description

The `SCOTCH_dgraphCoarsenVertLocMax` routine computes an upper bound on the number of coarse vertices that will be created locally when the given fine graph is coarsened (and possibly folded) by way of the `SCOTCH_dgraphCoarsen` routine (see Section 7.5.6). Its main use is to provide an upper bound

on the size of the `multloctab` multinode array that has to be pre-allocated before calling `SCOTCH_dgraphCoarsen`.

Because the number of local vertices of the coarse graph depends on the type of coarsening (*e.g.*, in case of folding, the number of local vertices of the coarse graph may be higher than that of the fine graph), the type of coarsening foreseen must be provided, using the `flagval` value. It must be the same value as the one that will be passed to the `SCOTCH_dgraphCoarsen` routine.

Return values

`SCOTCH_dgraphCoarsenVertLocMax` returns an upper bound on the local number of coarse vertices, which is henceforth always a non-negative number.

7.5.8 SCOTCH_dgraphData

Synopsis

```
void SCOTCH_dgraphData (const SCOTCH_Dgraph *  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 *             commptr)
```

```
scotchfdgraphdata (doubleprecision (*)  grafdat,
                   integer*num (*)      indxtab,
                   integer*num          baseval,
                   integer*num          vertglbnbr,
                   integer*num          vertlocnbr,
                   integer*num          vertlocmax,
                   integer*num          vertgstnbr,
                   integer*idx          vertlocidx,
                   integer*idx          vendlocidx,
                   integer*idx          velolocidx,
                   integer*idx          vlbllocidx,
                   integer*num          edgeglbnbr,
                   integer*num          edgelocnbr,
                   integer*num          edgelocsiz,
                   integer*idx          edgelocidx,
                   integer*idx          edgegstidx,
                   integer*idx          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`. `commptr` 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 `vertloctab` 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`. Also, on 32_64 architectures, such indices can be larger than the size of a regular `INTEGER`. This is why the indices to be returned are defined by means of a specific integer type. See Section 7.2.4 for more information on this issue.

7.5.9 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.

If `SCOTCH_dgraphInit` was called with a communicator that is not a predefined MPI communicator, it is the user's responsibility to free this communicator after all graphs that use it have been freed by means of the `SCOTCH_dgraphExit` routine.

7.5.10 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.

7.5.11 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.

7.5.12 SCOTCH_dgraphGrow

Synopsis

```
int SCOTCH_dgraphGrow (SCOTCH_Dgraph * const  grafptr,
                       const SCOTCH_Num       seedlocnbr,
                       SCOTCH_Num * const     seedloctab,
                       const SCOTCH_Num       distmax,
                       SCOTCH_Num * const     partgsttab)
```

```
scotchfdgraphgrow (doubleprecision (*)  grafdat,
                   integer*num          seedlocnbr,
                   integer*num (*)      seedloctab,
                   integer*num          distmax,
                   integer*num (*)      partgsttab,
                   integer              ierr)
```

Description

The SCOTCH_dgraphGrow routine grows areas in the SCOTCH_Dgraph structure pointed to by grafptr, from a set of colored seeds provided on each process. Several seeds can have the same color. Starting from the seeds declared in the initial local queues seedloctab on each process, at most distmax rounds of expansion will be performed, in a breadth-first search manner. During each round, all the neighbors of the vertices currently in queue are visited, and, if they have not already been colored, they receive the color of the current vertex and are in turn inserted into the queue.

The seedloctab array, which must be of a size at least equal to vertlocnbr as it will serve as the local vertex queue, holds the seedlocnbr local vertex indices of the local seeds. This represent the initial state of the vertex queue.

Colors are represented as positive or zero integer values. The partgsttab array, which must be of a size at least equal to vertgstnbr as it will also contain the colors of the ghost vertices received from neighboring processes, holds vertex colors. Its local part must be initialized with the colors of the seed vertices whose indices are contained in seedloctab. All its other local cells must be set to -1, in order for them to be traversed. Vertices not identified as seed vertices, and whose color is already set in partgsttab, will not be considered by the region growing process.

The values of vertlocnbr and vertgstnbr needed to dimension the arrays passed by the user can be queried by calling SCOTCH_dgraphData(), described in Section 7.5.8. The value of vertgstnbr will only be valid after SCOTCH_dgraphGhst() has been called on the given distributed graph (else, it will be equal to -1); see Section 7.7.1.

Return values

SCOTCH_dgraphGrow returns 0 if the region-growing process has succeeded, and 1 on error.

7.5.13 SCOTCH_dgraphInducePart

Synopsis

```
int SCOTCH_dgraphInducePart (SCOTCH_Dgraph * const      orggrafptr,
                             const SCOTCH_Num * const   orgpartloctab,
                             const SCOTCH_Num           indpartval,
                             const SCOTCH_Num           indvertlocnbr,
                             SCOTCH_Dgraph * const      indgrafptr)
```

```
scotchfdgraphinducepart (doubleprecision (*)  orggrafdat,
                         integer*num (*)      orgpartloctab,
                         integer*num          indpartval,
                         integer*num          indvertlocnbr,
                         doubleprecision (*)  indgrafdat,
                         integer              ierr)
```

Description

The SCOTCH_dgraphInducePart routine creates in the SCOTCH_Dgraph structure pointed to by indgrafptr a distributed induced subgraph of the SCOTCH_Dgraph pointed to by orggrafptr. The local vertices of every processor that are kept in the induced subgraph are the ones for which the values contained in the orgpartloctab array cells are equal to indpartval.

The orgpartloctab array must be of a size at least equal to the number of local vertices of the original graph. It may be larger, e.g. equal to the number of local plus ghost vertices, if needed by the user, but only the area corresponding to the local vertices will be used by SCOTCH_dgraphInducePart.

indvertlocnbr is the number of local vertices in the induced subgraph. It must therefore be equal to the number of local vertices that have their associated orgpartloctab cell value equal to indpartval. This value is necessary to internal array allocations. While it could have been easily computed by Scotch, by traversing the orgpartgsttab array, making it used-provided spares such a traversal if the user already knows the value. If it is not the case, setting this value to -1 will make Scotch compute it automatically.

indgrafptr must have been initialized with the SCOTCH_dgraphInit routine before SCOTCH_dgraphInducePart is called. The communicator that is passed to it can either be the communicator used by the original graph orggrafptr, or any congruent communicator created by using MPI_Comm_dup on this communicator. Using a distinct communicator for the induced subgraph allows subsequent library routines to be called in parallel on the two graphs after the induced graph is created.

Induced band graphs have vertex labels attached to each of their vertices, in the vlblloctab array. If the original graph had vertex labels attached to it, induced graph vertex labels are the labels of the corresponding vertices in the original graph. Else, induced graph vertex labels are the global indices of corresponding vertices in the original graph.

Depending on the partition array, the distribution of induced graph vertices may be highly imbalanced. In order for further computations on this distributed graph to scale well, a redistribution of its data may be necessary, using the SCOTCH_dgraphRedist routine.

Return values

SCOTCH_dgraphInducePart returns 0 if the induced graph structure has been successfully created, and 1 else.

7.5.14 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, the SCOTCH_dgraphExit function must be called, to free its internal data arrays.

If SCOTCH_dgraphInit was called with a communicator that is not a predefined MPI communicator (such as MPI_COMM_WORLD or MPI_COMM_SELF), it is the user's responsibility to free this communicator after all graphs that use it have been freed by means of the SCOTCH_dgraphExit routine.

Return values

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

7.5.15 SCOTCH_dgraphRedist

Synopsis

```
int SCOTCH_dgraphRedist (SCOTCH_Dgraph * const     orggrafptr,
                         const SCOTCH_Num * const  partloctab,
                         const SCOTCH_Num * const  permgsttab,
                         const SCOTCH_Num          vertlocdlt,
                         const SCOTCH_Num          edgelocdlt,
                         SCOTCH_Dgraph * const     redgrafptr)
```

```
scotchfdgraphredist (doubleprecision (*)  orggrafdat,
                     integer*num (*)      partloctab,
                     integer*num (*)      permgsttab,
                     integer*num          vertlocdlt,
                     integer*num          edgelocdlt,
                     doubleprecision (*)  redgrafptr)
```

Description

The SCOTCH_dgraphRedist routine initializes and fills the redistributed graph structure pointed to by redgrafptr with a new distributed graph made from data redistributed from the original graph pointed to by orggrafptr.

The partition array, partloctab, must always be provided. It holds the part number associated with each local vertex. Part indices are *not* based: target vertices are numbered from 0 to the number of parts minus 1.

Whenever provided, the permutation array permgsttab must be of a size equal to the number of local and ghost vertices of the source graph (that is, vertgstnbr, see Section 7.3.1). Its contents must be based, that is, permutation global indices start at baseval. Both its local and ghost contents must be valid. Consequently, it is the user's responsibility to call SCOTCH_dgraphHalo whenever necessary so as to propagate part values of the local vertices to their ghost counterparts on other processes. SCOTCH_dgraphRedist does not perform this halo exchange itself because users may already have computed these values by themselves when computing the new partition. If permgsttab is not provided by the user, vertices in each part are reordered according to their global indices in the source graph.

redgrafptr must have been initialized with the SCOTCH_dgraphInit routine before SCOTCH_dgraphRedist is called. The communicator that is passed to it can either be the communicator used by the original graph orggrafptr, or any congruent communicator created by using MPI_Comm_dup on this communicator. Using a distinct communicator for the redistributed graph allows subsequent library routines to be called in parallel on the two graphs after the redistributed graph is created.

Redistributed graphs have vertex labels attached to each of their vertices, in the vlblloctab array. If the original graph had vertex labels attached to it, redistributed graph vertex labels are the labels of the corresponding vertices in the original graph. Else, redistributed graph vertex labels are the global indices of corresponding vertices in the original graph.

Return values

SCOTCH_dgraphRedist returns 0 if the redistributed graph has been successfully created, and 1 else.

7.5.16 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.

7.5.17 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*num          vertglbnbr,
                   integer*num          vertlocnbr,
                   integer*num          edgeglbnbr,
                   integer*num          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.

7.5.18 SCOTCH_dgraphSizeof

Synopsis

```
int SCOTCH_dgraphSizeof (void)
scotchfdgraphsizeof (integer  size )
```

Description

The SCOTCH_dgraphSizeof routine returns the size, in bytes, of a SCOTCH_Dgraph structure. This information is useful to export the interface of the libPTScotch to interpreted languages, without access to the "scotch.h" include file.

7.5.19 SCOTCH_dgraphStat

Synopsis

```
int SCOTCH_dgraphStat (const SCOTCH_Graph *  grafptr,
                       SCOTCH_Num *          velominptr,
                       SCOTCH_Num *          velomaxptr,
                       SCOTCH_Num *          velosumptr,
                       double *              veloavgptr,
                       double *              velodltptr,
                       SCOTCH_Num *          degrminptr,
                       SCOTCH_Num *          degrmaxptr,
                       double *              degravgptr,
                       double *              degrdltptr,
                       SCOTCH_Num *          edlominptr,
                       SCOTCH_Num *          edlomaxptr,
                       SCOTCH_Num *          edlosumptr,
                       double *              edloavgptr,
                       double *              edlodltptr)
```

```
scotchfdgraphstat (doubleprecision (*)  grafdat,
                   integer*num          velomin,
                   integer*num          velomax,
                   integer*num          velosum,
                   doubleprecision      veloavg,
                   doubleprecision      velodlt,
                   integer*num          degrmin,
                   integer*num          degrmax,
                   doubleprecision      degravg,
                   doubleprecision      degrdlt,
                   integer*num          edlomin,
                   integer*num          edlomax,
                   integer*num          edlosum,
                   doubleprecision      edloavg,
                   doubleprecision      edlodlt,
                   integer              ierr)
```

Description

The `SCOTCH_dgraphStat` routine produces some statistics regarding the distributed graph structure pointed to by `grafptr`. `velomin`, `velomax`, `velosum`, `veloavg` and `velodlt` are the minimum vertex load, the maximum vertex load, the sum of all vertex loads, the average vertex load, and the variance of the vertex loads, respectively. `degrmin`, `degrmax`, `degravg` and `degrdlt` are the minimum vertex degree, the maximum vertex degree, the average vertex degree, and the variance of the vertex degrees, respectively. `edlomin`, `edlomax`, `edlosum`, `edloavg` and `edlodlt` are the minimum edge load, the maximum edge load, the sum of all edge loads, the average edge load, and the variance of the edge loads, respectively.

Return values

`SCOTCH_dgraphStat` returns 0 if the statistics have been successfully produced, and 1 else.

7.6 Distributed graph I/O routines

7.6.1 SCOTCH_dgraphLoad

Synopsis

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

scotchfdgraphload (doubleprecision (*)  grafdat,
                   integer              fildes,
                   integer*num          baseval,
                   integer*num          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 5.1 for a description of the distributed graph format, and to the Scotch *User's Guide* [35] 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 PXFFILENO or FNUM functions 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.

7.6.2 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 5.1).

Fortran users must use the PXFFILENO or FNUM functions 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.

7.7 Data handling and exchange routines

7.7.1 SCOTCH_dgraphGhst

Synopsis

```
int SCOTCH_dgraphGhst (SCOTCH_Dgraph * const  grafptr)

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

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 7.3.1.

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.

7.7.2 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, in a synchronous way. 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 [32, 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, if it was not already.

In case useful computation can be carried out during the halo exchange, an asynchronous version of this routine is available, called SCOTCH_dgraphHalo Async.

Return values

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

7.7.3 SCOTCH_dgraphHaloAsync

Synopsis

```
int SCOTCH_dgraphHaloAsync (SCOTCH_Dgraph * const         grafptr,
                            void *                        datatab,
                            MPI_Datatype                  typeval,
                            SCOTCH_DgraphHaloReq * const  requptr)

scotchfdgraphhaloasync (doubleprecision (*)  grafdat,
                        doubleprecision (*)  datatab,
                        integer              typeval,
                        doubleprecision (*)  requptr,
                        integer              ierr)
```

Description

The SCOTCH_dgraphHaloAsync routine propagates the data borne by local vertices to all of the corresponding halo vertices located on neighboring processes, in an asynchronous way. 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.

The semantics of SCOTCH_dgraphHaloAsync is similar to that of SCOTCH_dgraphHalo, except that it returns as soon as possible, while effective communication may not have started nor completed. Also, it possesses an additional parameter, requptr, which must point to a SCOTCH_DgraphHaloReq data structure. Similarly to asynchronous MPI calls, users can wait for the completion of a SCOTCH_dgraphHaloAsync routine by calling the SCOTCH_dgraphHaloWait routine, passing it a pointer to this request structure. In Fortran, the request structure must be defined as an array of DOUBLEPRECISIONs, of size SCOTCH_DGRAPHHALOREQDIM. This constant is defined in file ptscotchf.h, which must be included whenever necessary.

The effective means for SCOTCH_dgraphHaloAsync to perform its task may vary at compile time, depending on the presence of a thread-safe MPI library or on the existence of asynchronous collective communication routines such as MPE_Ialltoallv. In case no method for performing asynchronous collective communication is available, SCOTCH_dgraphHaloAsync will internally call SCOTCH_dgraphHalo to perform synchronous communication.

Because of possible limitations in the implementation of third-party communication routines, it is not recommended to perform simultaneous SCOTCH_dgraphHaloAsync calls on the same communicator.

Return values

SCOTCH_dgraphHaloAsync returns 0 if the halo data exchange has been successfully started, and 1 else.

7.7.4 SCOTCH_dgraphHaloWait

Synopsis

```
int SCOTCH_dgraphHaloWait (SCOTCH_DgraphHaloReq * const  requptr)

scotchfdgraphhalowait (doubleprecision (*)  requptr,
                       integer              ierr)
```

Description

The SCOTCH_dgraphHaloWait routine waits for the termination of an asynchronous halo exchange process, started by a call to SCOTCH_dgraphHalo Async, and represented by its request, pointed to by requptr.

In Fortran, the request structure must be defined as an array of DOUBLEPRECISIONs, of size SCOTCH_DGRAPHHALOREQDIM. This constant is defined in file ptscotchf.h, which must be included whenever necessary.

Return values

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

7.8 Distributed graph mapping and partitioning routines

SCOTCH_dgraphMap and SCOTCH_dgraphPart provide high-level functionalities and free the user from the burden of calling in sequence several of the low-level routines also described in this section.

7.8.1 SCOTCH_dgraphMap

Synopsis

```
int SCOTCH_dgraphMap (const SCOTCH_Dgraph *  grafptr,
                      const SCOTCH_Arch *    archptr,
                      const SCOTCH_Strat *   straptr,
                      SCOTCH_Num *           partloctab)

scotchfdgraphmap (doubleprecision (*)  grafdat,
                  doubleprecision (*)  archdat,
                  doubleprecision (*)  stradat,
                  integer*num (*)      partloctab,
                  integer              ierr)
```

Description

The SCOTCH_dgraphMap routine computes a mapping of the distributed source graph structure pointed to by grafptr onto the target architecture pointed to by archptr, using the mapping strategy pointed to by straptr, and returns distributed fragments of the partition data in the array pointed to by partloctab.

The partloctab array should have been previously allocated, of a size sufficient to hold as many SCOTCH_Num integers as there are local vertices of the source graph on each of the processes.

On return, every cell of the mapping array holds the number of the target vertex to which the corresponding source vertex is mapped. The numbering of target values is *not* based: target vertices are numbered from 0 to the number of target vertices minus 1.

Attention: version 6.0 of Scotch does not allow yet to map distributed graphs onto target architectures which are not complete graphs. This restriction will be removed in the next release.

Return values

SCOTCH_dgraphMap returns 0 if the partition of the graph has been successfully computed, and 1 else. In this last case, the partloctab arrays may however have been partially or completely filled, but their contents is not significant.

7.8.2 SCOTCH_dgraphMapCompute

Synopsis

```
int SCOTCH_dgraphMapCompute (const SCOTCH_Dgraph *  grafptr,
                             SCOTCH_Dmapping *      mappptr,
                             const SCOTCH_Strat *   straptr)

scotchfdgraphmapcompute (doubleprecision (*)  grafdat,
                         doubleprecision (*)  mappdat,
                         doubleprecision (*)  stradat,
                         integer              ierr)
```

Description

The SCOTCH_dgraphMapCompute routine computes a mapping on the given SCOTCH_Dmapping structure pointed to by mappptr using the parallel mapping strategy pointed to by straptr.

On return, every cell of the distributed mapping array (see Section 7.8.4) holds the number of the target vertex to which the corresponding source vertex is mapped. The numbering of target values is *not* based: target vertices are numbered from 0 to the number of target vertices, minus 1.

Attention: version 6.0 of Scotch does not allow yet to map distributed graphs onto target architectures which are not complete graphs. This restriction will be removed in the next release.

Return values

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

7.8.3 SCOTCH_dgraphMapExit

Synopsis

```
void SCOTCH_dgraphMapExit (const SCOTCH_Dgraph *  grafptr,
                           SCOTCH_Dmapping *      mappptr)

scotchfdgraphmapexit (doubleprecision (*)  grafdat,
                      doubleprecision (*)  mappdat)
```

Description

The SCOTCH_dgraphMapExit function frees the contents of a SCOTCH_Dmapping structure previously initialized by SCOTCH_dgraphMapInit. All subsequent calls to SCOTCH_dgraphMap* routines other than SCOTCH_dgraphMapInit, using this structure as parameter, may yield unpredictable results.

7.8.4 SCOTCH_dgraphMapInit

Synopsis

```
int SCOTCH_dgraphMapInit (const SCOTCH_Dgraph *  grafptr,
                          SCOTCH_Dmapping *      mappptr,
                          const SCOTCH_Arch *    archptr,
                          SCOTCH_Num *           partloctab)

scotchfdgraphmapinit (doubleprecision (*)  grafdat,
                      doubleprecision (*)  mappdat,
                      doubleprecision (*)  archdat,
                      integer*num (*)      partloctab,
                      integer              ierr)
```

Description

The SCOTCH_dgraphMapInit routine fills the distributed mapping structure pointed to by mappptr with all of the data that is passed to it. Thus, all subsequent calls to ordering routines such as SCOTCH_dgraphMapCompute, using this mapping structure as parameter, will place mapping results in field partloctab.

partloctab is the pointer to an array of as many SCOTCH_Nums as there are local vertices in each local fragment of the distributed graph pointed to by grafptr, and which will receive the indices of the vertices of the target architecture pointed to by archptr.

It should be the first function to be called upon a SCOTCH_Dmapping structure. When the distributed mapping structure is no longer of use, call function SCOTCH_dgraphMapExit to free its internal structures.

Return values

SCOTCH_dgraphMapInit returns 0 if the distributed mapping structure has been successfully initialized, and 1 else.

7.8.5 SCOTCH_dgraphMapSave

Synopsis

```
int SCOTCH_dgraphMapSave (const SCOTCH_Dgraph *    grafptr,
                          const SCOTCH_Dmapping *  mappptr,
                          FILE *                   stream)

scotchfdgraphmapsave (doubleprecision (*)  grafdat,
                      doubleprecision (*)  mappdat,
                      integer              fildes,
                      integer              ierr)
```

Description

The SCOTCH_dgraphMapSave routine saves the contents of the SCOTCH_Dmapping structure pointed to by mappptr to stream stream, in the Scotch mapping format. Please refer to the Scotch *User's Guide* [35] for more information about this format.

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

Fortran users must use the PXFFILENO or FNUM functions to obtain the number of the Unix file descriptor fildes associated with the logical unit of the mapping file.

Return values

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

7.8.6 SCOTCH_dgraphMapStat

Synopsis

```
int SCOTCH_dgraphMapStat (SCOTCH_Dgraph * const          grafptr,
                          const SCOTCH_Dmapping * const  mappptr,
                          SCOTCH_Num * const             tgtnbrptr,
                          SCOTCH_Num * const             mapnbrptr,
                          SCOTCH_Num * const             mapminptr,
                          SCOTCH_Num * const             mapmaxptr,
                          double * const                 mapavgptr,
                          double * const                 mapdltptr,
                          SCOTCH_Num * const             ngbsumptr,
                          SCOTCH_Num * const             ngbminptr,
                          SCOTCH_Num * const             ngbmaxptr,
                          SCOTCH_Num * const             cdstmaxptr,
                          SCOTCH_Num                     cdsttab[256],
                          SCOTCH_Num * const             cmlosumptr,
                          SCOTCH_Num * const             cmdisumptr,
                          SCOTCH_Num * const             cmexsumptr)
```

```
scotchfdgraphmapstat (doubleprecision (*)  grafdat,
                      doubleprecision (*)  mappdat,
                      integer*num          tgtnbr,
                      integer*num          mapnbr,
                      integer*num          mapmax,
                      integer*num          mapmin,
                      doubleprecision      mapavg,
                      doubleprecision      mapdlt,
                      integer*num          ngbsum,
                      integer*num          ngbmin,
                      integer*num          ngbmax,
                      integer*num          cdstmax,
                      integer*num (*)      cdsttab,
                      integer*num          cmlosum,
                      integer*num          cmdisum,
                      integer*num          cmexsum,
                      integer              ierr)
```

Description

The `SCOTCH_dgraphMapStat` routine computes a set of statistics on the distributed mapping structure pointed to by `mappptr`, resuting from the mapping of the distributed graph pointed to by `grafdat`.

A subdomain of a mapping is the set of all the source graph vertices mapped onto the same target vertex. Source graph vertices that are not associated with any target vertex are said to be unmapped.

Upon successful return, all variables passed as references will contain the relevant statistics values. When a value is not needed, the user may pass a `NULL` pointer instead.

`tgtnbrptr` will hold the number of vertices in the target architecture used for mapping, and `mapnbrptr` will hold the number of target vertices actually used by the mapping, which may be smaller. `mapminptr` and `mapmaxptr` will hold the minimum and maximum loads mapped onto the target vertices used, with respect to their weights, and `mapavgptr` and `mapdltptr` will hold the average and standard deviation of the loads assigned to the target vertices used, also with respect to their weights.

`ngbsumptr` will hold the sum of the number of neighboring subdomains, computed over all subdomains of the mapping, while `ngbminptr` and `ngbmaxptr` will hold the minimum and maximum numbers of neighboring subdomains, respectively.

`cmlosumptr` will hold the overall communication load in the considered mapping, *i.e.*, the sum of the weights of all edges in the source graph whose two ends are mapped. Consequently, this number may be smaller than the sum of all the weights of the source graph edges. `cmdisumptr` will hold the total communication dilation, *i.e.*, the sum of the distances, in the target architecture, between the two ends of all the mapped edges. `cmexsumptr` will hold the total communication expansion, *i.e.*, the sum of the distances, in the target architecture, between the two ends of all the mapped edges, multiplied by their edge weight. See Section 3.1.2 for an explanation on these values.

Finally, the `cdsttab` array will hold the breakdown, by communication distance (capped to 255), of the communication loads of mapped edges. Hence, `cdsttab[0]` will contain the sum of uncut edge loads, *i.e.*, local communication. `cdstmaxptr` will hold the maximum edge dilation in the target graph (capped to 255), *i.e.*, the highest index of non-zero cells in `cdsttab`.

Return values

`SCOTCH_dgraphMapStat` returns 0 if the statistics have been successfully computed, and 1 else.

7.8.7 `SCOTCH_dgraphMapView`

Synopsis

```
int SCOTCH_dgraphMapView (const SCOTCH_Dgraph *    grafptr,
                          const SCOTCH_Dmapping *  mappptr,
                          FILE *                   stream)

scotchfdgraphmapview (doubleprecision (*)  grafdat,
                      doubleprecision (*)  mappdat,
                      integer              fildes,
                      integer              ierr)
```

Description

The `SCOTCH_dgraphMapView` routine summarizes statistical information on the mapping pointed to by `mappptr` (load of target processors, number of neighboring domains, average dilation and expansion, edge cut size, distribution of edge dilations), and prints these results to stream `stream`.

Fortran users must use the `PXFFILENO` or `FNUM` functions to obtain the number of the Unix file descriptor `fildes` associated with the logical unit of the output data file.

Return values

`SCOTCH_dgraphMapView` returns 0 if the data has been successfully written to `stream`, and 1 else.

7.8.8 `SCOTCH_dgraphPart`

Synopsis

```
int SCOTCH_dgraphPart (const SCOTCH_Dgraph *  grafptr,
                       const SCOTCH_Num       partnbr,
                       const SCOTCH_Strat *   straptr,
                       SCOTCH_Num *           partloctab)

scotchfdgraphpart (doubleprecision (*)  grafdat,
                   integer*num          partnbr,
                   doubleprecision (*)  stradat,
                   integer*num (*)      partloctab,
                   integer              ierr)
```

Description

The SCOTCH_dgraphPart routine computes a partition into partnbr parts of the distributed source graph structure pointed to by grafptr, using the graph partitioning strategy pointed to by straptr, and returns distributed fragments of the partition data in the array pointed to by partloctab.

The partloctab array should have been previously allocated, of a size sufficient to hold as many SCOTCH_Num integers as there are local vertices of the source graph on each of the processes.

On return, every array cell holds the number of the part to which the corresponding vertex is mapped. Parts are numbered from 0 to partnbr − 1.

Return values

SCOTCH_dgraphPart returns 0 if the partition of the graph has been successfully computed, and 1 else. In this latter case, the partloctab array may however have been partially or completely filled, but its content is not significant.

7.9 Distributed graph ordering routines

7.9.1 SCOTCH_dgraphOrderCblkDist

Synopsis

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

scotchfdgraphordercblkdist (doubleprecision (*)  grafdat,
                            doubleprecision (*)  ordedat,
                            integer*num          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.

7.9.2 SCOTCH_dgraphOrderCompute

Synopsis

```
int SCOTCH_dgraphOrderCompute (SCOTCH_Dgraph * const     grafptr,
                               SCOTCH_Ordering * const   ordeptr,
                               SCOTCH_Strat * const      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 straptr, 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.

7.9.3 SCOTCH_dgraphOrderComputeList

Synopsis

```
int SCOTCH_dgraphOrderComputeList (SCOTCH_Dgraph *     grafptr,
                                   SCOTCH_Ordering *   ordeptr,
                                   SCOTCH_Num          listnbr,
                                   const SCOTCH_Num *  listtab,
                                   SCOTCH_Strat *      straptr)

scotchfdgraphordercomputelist (doubleprecision (*)  grafdat,
                               doubleprecision (*)  ordedat,
                               integer*num          listnbr,
                               integer*num (*)      listtab,
                               doubleprecision (*)  stradat,
                               integer              ierr)
```

Description

The SCOTCH_dgraphOrderComputeList routine computes in parallel a distributed block ordering of a subgraph of the graph structure pointed to by grafptr, using the distributed ordering strategy pointed to by straptr, and stores its result in the distributed ordering structure pointed to by ordeptr. The induced subgraph is described by means of local vertex lists, one for each process: listnbr holds the number of local vertices to keep in the induced

subgraph, the local indices of which are given, in any order, in the `listtab` array. Note that `listtab` must contain local vertex numbers, i.e ranging from `baseval` to `vertlocnbr` + `baseval`.

Because an ordering always refers to the full graph, the ordering computed by `SCOTCH_dgraphOrderComputeList` is divided into two disconnected components: the induced graph vertices are ordered by applying the strategy provided by the `straptr` parameter, while vertices excluded from the list are ordered consecutively with the highest available indices. Consequently, the permuted indices of induced vertices range from `baseval` to (`listnbr` + `baseval` − 1), while the permuted indices of the remaining vertices range from (`listnbr` + `baseval`) to (`vertnbr` + `baseval` − 1), inclusive. Accordingly, the separation tree yielded by `SCOTCH_dgraphOrderComputeList` starts from a root with two children, the first one corresponding to the induced subgraph, and the second one to the vertices excluded from the list.

Return values

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

7.9.4 `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.

7.9.5 `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.

7.9.6 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* [35] 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 PXFFILENO or FNUM functions 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.

7.9.7 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 PXFFILENO or FNUM functions 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.

7.9.8 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 that 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 `PXFFILENO` or `FNUM` functions 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.

7.9.9 SCOTCH_dgraphOrderPerm

Synopsis

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

scotchfdgraphorderperm (doubleprecision (*)  grafdat,
                        doubleprecision (*)  ordedat,
                        integer*num (*)      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 must be of size `vertlocnbr`.

Return values

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

7.9.10 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*num (*)      treeglbtab,
                            integer*num (*)      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, a disconnected components node with two sons as computed by SCOTCH_dgraphOrderComputeList (see Section 7.9.3), 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 that 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_Nums 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.

7.10 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.

7.10.1 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`.

7.10.2 `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*num (*)      permtab,
                         integer*num (*)      peritab,
                         integer*num          cblknbr,
                         integer*num (*)      rangtab,
                         integer*num (*)      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 onto 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 7.3.3 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.

7.10.3 SCOTCH_dgraphOrderGather

Synopsis

```
int SCOTCH_dgraphOrderGather (const SCOTCH_Dgraph *  grafptr,
                              SCOTCH_Dordering *     dordptr,
                              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.

Only the process onto which distributed ordering data is to be centralized has to provide a centralized ordering structure, initialized by way of the SCOTCH_dgraphCorderInit routine. Other processes have to pass a NULL pointer for the cordptr parameter, or provide the dorddat structure as the corddat parameter in Fortran.

Return values

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

7.11 Distributed mesh handling routines

7.11.1 SCOTCH_dmeshAlloc

Synopsis

```
SCOTCH_Dmesh * SCOTCH_dmeshAlloc (void)
```

Description

The SCOTCH_dmeshAlloc function allocates a memory area of a size sufficient to store a SCOTCH_Dmesh structure. It is the user's responsibility to free this memory when it is no longer needed. The allocated space must be initialized before use, by means of the SCOTCH_dmeshInit routine.

Return values

SCOTCH_dmeshAlloc returns the pointer to the memory area if it has been successfully allocated, and NULL else.

7.11.2 SCOTCH_dmeshBuildAdm

Synopsis

```
int SCOTCH_dmeshBuildAdm (SCOTCH_Dmesh *      meshptr,
                          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)

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

Description

The SCOTCH_dmeshBuildAdm routine fills the distributed source mesh structure pointed to by meshptr with all of the data that are passed to it in the auxiliary distributed mesh format (see Section 5.2 for more information on this format).

baseval is the mesh base value for index arrays (typically 0 for structures built from C and 1 for structures built from Fortran). velmlocnbr is the number of local element vertices on the calling process. velmloctab is the local element-to-node adjacency index array, of size (velmlocnbr + 1). eelmlocnbr is the

local number of element-to-node arcs. `eelmloctab` is the local adjacency array, of size `eelmlocnbr`, which stores the global indices of end node vertices. `vnodglbnbr` is the global number of node vertices. All these data must be provided on all processes.

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

Return values

`SCOTCH_dmeshBuildAdm` returns 0 if the mesh structure has been successfully set with all of the input data, and 1 else.

7.11.3 `SCOTCH_dmeshData`

Synopsis

```
void SCOTCH_dmeshData (const SCOTCH_Dmesh *  meshptr,
                       SCOTCH_Num *          baseptr,
                       SCOTCH_Num *          velmglbptr,
                       SCOTCH_Num *          velmlocptr,
                       SCOTCH_Num **         velmloctab,
                       SCOTCH_Num *          eelmglbnbr,
                       SCOTCH_Num *          eelmlocptr,
                       SCOTCH_Num **         eelmloctab,
                       SCOTCH_Num *          vnodglbptr,
                       MPI_Comm *            commptr)

scotchfdmeshdata (doubleprecision (*)  meshdat,
                  integer*num (*)      indxtab,
                  integer*num          baseval,
                  integer*num          velmglbnbr,
                  integer*num          velmlocnbr,
                  integer*idx          velmlocidx,
                  integer*num          eelmglbnbr,
                  integer*num          eelmlocnbr,
                  integer*idx          eelmlocidx,
                  integer*num          vnodglbnbr,
                  integer              comm)
```

Description

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

`baseptr` is the pointer to a location that will hold the mesh base value for index arrays (typically 0 for structures built from C and 1 for structures built

from Fortran). velmglbptr is the pointer to a location that will hold the global number of element vertices. velmlocptr is the pointer to a location that will hold the number of local element vertices. vertloctab is the pointer to a location that will hold the reference to the adjacency index array, of size *velmlocptr + 1. eelmglbptr is the pointer to a location that will hold the global number of element-to-node arcs, which is also the global number of node-to-element arcs. eelmlocptr is the pointer to a location that will hold the number of local element-to-node arcs. eelmloctab is the pointer to a location that will hold the reference to the local element-to-node adjacency array of global indices, of size *eelmlocptr. vnodglbptr is the pointer to a location that will hold the global number of node vertices. commptr is the pointer to a location that will hold the MPI communicator of the distributed mesh.

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 mesh arrays. The scotchfdmeshdata 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 velmlocidx, the index of velmloctab, if they do not exist. For instance, if some base array myarray (1) is passed as parameter indxtab, then the first cell of array velmloc tab will be accessible as myarray(velmlocidx). In order for this feature to behave properly, the indxtab array must be word-aligned with the mesh 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. Also, on 32_64 architectures, such indices can be larger than the size of a regular INTEGER. This is why the indices to be returned are defined by means of a specific integer type. See Section 7.2.4 for more information on this issue.

7.11.4 SCOTCH_dmeshDgraphDual

Synopsis

```
int SCOTCH_dmeshDgraphDual (const SCOTCH_Dmesh *  meshptr,
                            SCOTCH_Dgraph *       grafptr,
                            const SCOTCH_Num      nocoval)

scotchfdmeshdgraphdual (doubleprecision (*)  meshdat,
                        doubleprecision (*)  grafdat,
                        integer              nocoval,
                        integer              ierr)
```

Description

The SCOTCH_dmeshDgraphDual routine builds a distributed dual graph (*i.e.*, an element graph) from a distributed mesh. It creates, in the SCOTCH_Dgraph structure pointed to by grafptr, a distributed graph having as many vertices as there are elements in the SCOTCH_Dmesh structure pointed to by meshptr, and such that there exists an edge between any two graph vertices if there are at least nocoval shared nodes between the two corresponding elements in the source mesh.

Return values

SCOTCH_dmeshDgraphDual returns 0 if the distributed graph structure has been successfully allocated and filled, and 1 else.

7.11.5 SCOTCH_dmeshExit

Synopsis

```
void SCOTCH_dmeshExit (SCOTCH_Dmesh *  meshptr)
scotchfdmeshexit (doubleprecision (*)  meshdat)
```

Description

The SCOTCH_dmeshExit function frees the contents of a SCOTCH_Dmesh structure previously initialized by SCOTCH_dmeshInit. All subsequent calls to SCOTCH_mesh routines other than SCOTCH_dmeshInit, using this structure as parameter, may yield unpredictable results.

If SCOTCH_dmeshInit was called with a communicator that is not a predefined MPI communicator, it is the user's responsibility to free this communicator after all meshes that use it have been freed by means of the SCOTCH_dmesh Exit routine.

7.11.6 SCOTCH_dmeshFree

Synopsis

```
void SCOTCH_dmeshFree (SCOTCH_Dmesh *  meshptr)
scotchfdmeshfree (doubleprecision (*)  meshdat)
```

Description

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

7.11.7 SCOTCH_dmeshInit

Synopsis

```
int SCOTCH_dmeshInit (SCOTCH_Dmesh *  meshptr,
                      MPI_Comm        comm)

scotchfdmeshinit (doubleprecision (*)  meshdat,
                  integer              comm,
                  integer              ierr)
```

Description

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

SCOTCH_dmeshInit 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 mesh 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 mesh is no longer of use, the SCOTCH_dmeshExit function must be called, to free its internal data arrays.

If SCOTCH_dmeshInit was called with a communicator that is not a predefined MPI communicator (such as MPI_COMM_WORLD or MPI_COMM_SELF), it is the user's responsibility to free this communicator after all meshes that use it have been freed by means of the SCOTCH_dmeshExit routine.

Return values

SCOTCH_dmeshInit returns 0 if the gmesh structure has been successfully initialized, and 1 else.

7.11.8 SCOTCH_dmeshSize

Synopsis

```
void SCOTCH_dmeshSize (const SCOTCH_Dmesh *  meshptr,
                       SCOTCH_Num *          velmglbptr,
                       SCOTCH_Num *          velmlocptr,
                       SCOTCH_Num *          eelmglbnbr,
                       SCOTCH_Num *          eelmlocptr,
                       SCOTCH_Num *          vnodglbptr)
```

```
scotchfdmeshsize (doubleprecision (*)  meshdat,
                  integer*num          velmglbnbr,
                  integer*num          velmlocnbr,
                  integer*num          eelmglbnbr,
                  integer*num          eelmlocnbr,
                  integer*num          vnodglbnbr)
```

Description

The SCOTCH_dmeshSize routine fills the five areas of type SCOTCH_Num pointed to by velmglbptr, velmlocptr, eelmglbptr, eelmlocptr and vnodglbptr with the number of global element vertices and element-to-node arcs of the given mesh pointed to by meshptr, as well as with the number of local element vertices and element-to-node arcs borne by each of the calling processes, and the global number of node vertices.

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 mesh read by means of the SCOTCH_dmeshLoad routine, in order to allocate auxiliary arrays of proper sizes. If the whole structure of the mesh is wanted, function SCOTCH_dmeshData should be preferred.

7.11.9 SCOTCH_dmeshSizeof

Synopsis

```
int SCOTCH_dmeshSizeof (void)
scotchfdmeshsizeof (integer  size )
```

Description

The SCOTCH_dmeshSizeof routine returns the size, in bytes, of a SCOTCH_Dmesh structure. This information is useful to export the interface of the libPTScotch to interpreted languages, without access to the "scotch.h" include file.

7.12 Distributed mesh I/O routines

7.12.1 SCOTCH_dmeshLoad

Synopsis

```
int SCOTCH_dmeshLoad (SCOTCH_Dmesh *  meshptr,
                      FILE *          stream,
                      SCOTCH_Num      baseval,
                      SCOTCH_Num      flagval)
```

```
scotchfdmeshload (doubleprecision (*)  meshdat,
                  integer              fildes,
                  integer*num          baseval,
                  integer*num          flagval,
                  integer              ierr)
```

Description

The `SCOTCH_dmeshLoad` routine fills the `SCOTCH_Dmesh` structure pointed to by `meshptr` with the distributed source mesh description available from multiple streams `stream` in the Scotch distributed mesh format (please refer to Section 5.2 for a description of the auxiliary distributed mesh format).

To ease the handling of source mesh files by programs written in C as well as in Fortran, the base value of the mesh 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 mesh base should be the same as the one provided in the mesh description that is read from `stream`.

The `flagval` value is currently not used. This value should be set to `0`.

Fortran users must use the `PXFFILENO` or `FNUM` functions to obtain the number of the Unix file descriptor `fildes` associated with the logical unit of the mesh file.

Return values

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

7.13 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.

7.13.1 `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.

7.13.2 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.

7.13.3 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 PXFFILENO or FNUM functions 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.

7.14 Strategy creation routines

Strategy creation routines parse the user-provided strategy string and populate the given opaque strategy object with a tree-shaped structure that represents the parsed

expression. It is this structure that will be later traversed by the generic routines for partitioning, mapping or ordering, so as to determine which specific partitioning, mapping or ordering method to be called on a subgraph being considered.

Because strategy creation routines call third-party lexical analyzers that may have been implemented in a non-reentrant way, no guarantee is given on the reentrance of these routines. Consequently, strategy creation routines that might be called simultaneously by multiple threads should be protected by a mutex.

7.14.1 SCOTCH_stratDgraphClusterBuild

Synopsis

```
int SCOTCH_stratDgraphClusterBuild (SCOTCH_Strat *    straptr,
                                    const SCOTCH_Num  flagval,
                                    const SCOTCH_Num  procnbr,
                                    const SCOTCH_Num  pwgtmax,
                                    const double      densmin,
                                    const double      bbalval)

scotchfstratdgraphclusterbuild (doubleprecision (*)  stradat,
                                integer*num          flagval,
                                integer*num          procnbr,
                                integer*num          pwgtmax,
                                doubleprecision      densmin,
                                doubleprecision      bbalval,
                                integer              ierr)
```

Description

The SCOTCH_stratDgraphClusterBuild routine fills the strategy structure pointed to by straptr with a default clustering strategy tuned according to the preference flags passed as flagval, the maximum cluster vertex weight pwgtmax, the minimum edge density densmin, and the bipartition imbalance ratio bbalval, to be used on procnbr processes.

From this point, the strategy structure can only be used as a mapping strategy, to be used by a mapping function such as SCOTCH_graphMap.

Recursive bipartitioning will be applied to the graph, every bipartition allowing for an imbalance tolerance of bbalval. Recursion will stop if either cluster size becomes smaller than pwgtmax, or cluster edge density becomes higher than densmin, which represents the fraction of edges internal to the cluster with respect to a complete graph. See Section 7.4.1 for a description of the available flags.

Return values

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

7.14.2 SCOTCH_stratDgraphMap

Synopsis

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

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

Description

The SCOTCH_stratDgraphMap routine fills the strategy structure pointed to by straptr with the distributed graph mapping strategy string pointed to by string. The format of this strategy string is described in Section 7.4.2. From this point, strategy strat can only be used as a distributed graph mapping strategy, to be used by functions SCOTCH_dgraphPart, SCOTCH_dgraphMap or SCOTCH_dgraphMapCompute. 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_stratDgraphMap returns 0 if the strategy string has been successfully set, and 1 else.

7.14.3 SCOTCH_stratDgraphMapBuild

Synopsis

```
int SCOTCH_stratDgraphMapBuild (SCOTCH_Strat *    straptr,
                                const SCOTCH_Num  flagval,
                                const SCOTCH_Num  procnbr,
                                const SCOTCH_Num  partnbr,
                                const double      balrat)

scotchfstratdgraphmapbuild (doubleprecision (*)  stradat,
                            integer*num          flagval,
                            integer*num          procnbr,
                            integer*num          partnbr,
                            doubleprecision      balrat,
                            integer              ierr)
```

Description

The SCOTCH_stratDgraphMapBuild routine fills the strategy structure pointed to by straptr with a default mapping strategy tuned according to the preference flags passed as flagval and to the desired number of parts partnbr and imbalance ratio balrat, to be used on procnbr processes. From this point,

the strategy structure can only be used as a parallel mapping strategy, to be used by function SCOTCH_dgraphMap, for instance. See Section 7.4.1 for a description of the available flags.

Return values

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

7.14.4 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. The format of this strategy string is described in Section 7.4.4. 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.

7.14.5 SCOTCH_stratDgraphOrderBuild

Synopsis

```
int SCOTCH_stratDgraphOrderBuild (SCOTCH_Strat *    straptr,
                                  const SCOTCH_Num  flagval,
                                  const SCOTCH_Num  procnbr,
                                  const SCOTCH_Num  levlnbr,
                                  const double      balrat)

scotchfstratdgraphorderbuild (doubleprecision (*)  stradat,
                              integer*num           flagval,
                              integer*num           procnbr,
                              integer*num           levlnbr,
                              doubleprecision       balrat,
                              integer               ierr)
```

Description

The SCOTCH_stratDgraphOrderBuild routine fills the strategy structure pointed to by straptr with a default parallel ordering strategy tuned according to the preference flags passed as flagval and to the desired nested dissection imbalance ratio balrat, to be used on procnbr processes. From this point, the strategy structure can only be used as a parallel ordering strategy, to be used by function SCOTCH_dgraphOrder, for instance.

See Section 7.4.1 for a description of the available flags. When any of the SCOTCH_STRATLEVELMIN or SCOTCH_STRATLEVELMAX flags is set, the levlnbr parameter is taken into account.

Return values

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

7.15 Other data structure routines

7.15.1 SCOTCH_dmapAlloc

Synopsis

```
SCOTCH_Dmapping * SCOTCH_dmapAlloc (void)
```

Description

The SCOTCH_dmapAlloc function allocates a memory area of a size sufficient to store a SCOTCH_Dmapping structure. It is the user's responsibility to free this memory when it is no longer needed.

Return values

SCOTCH_dmapAlloc returns the pointer to the memory area if it has been successfully allocated, and NULL else.

7.15.2 SCOTCH_dmapSizeof

Synopsis

```
int SCOTCH_dmapSizeof (void)
scotchfdmapsizeof (integer  size )
```

Description

The SCOTCH_dmapSizeof routine returns the size, in bytes, of a SCOTCH_Dmapping structure. This information is useful to export the interface of the libPTScotch to interpreted languages, without access to the "scotch.h" include file.

7.15.3 SCOTCH_dorderAlloc

Synopsis

```
SCOTCH_Dordering * SCOTCH_dorderAlloc (void)
```

Description

The SCOTCH_dorderAlloc function allocates a memory area of a size sufficient to store a SCOTCH_Dordering structure. It is the user's responsibility to free this memory when it is no longer needed.

Return values

SCOTCH_dorderAlloc returns the pointer to the memory area if it has been successfully allocated, and NULL else.

7.15.4 SCOTCH_dorderSizeof

Synopsis

```
int SCOTCH_dorderSizeof (void)
scotchfdordersizeof (integer  size )
```

Description

The SCOTCH_dorderSizeof routine returns the size, in bytes, of a SCOTCH_Dordering structure. This information is useful to export the interface of the libPTScotch to interpreted languages, without access to the "scotch.h" include file.

7.16 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.

7.16.1 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.

7.16.2 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.

7.16.3 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.

7.17 Random generator handling

In order not to be influenced by the concurrent execution of third-party software and/or library routines, the libScotch library embeds its own pseudo-random number generator. This generator is used by default by all libScotch routines.

When Scotch has been compiled with any of the flags COMMON_RANDOM_FIXED_SEED or SCOTCH_DETERMINISTIC set, this random number generator is initialized with a prescribed, default seed. In this case, any two runs of the same sequence of libScotch routines will yield the same result. The first flag will be sufficient when Scotch is run on a single thread, while he second one is necessary when Scotch is run on several threads, because multi-threaded versions of the libScotch routines

may rely by default on non-deterministic algorithms that are not only sensitive to the pseudo-random sequence but also to system artifacts (see Section 7.1).

In certain cases, it may be interesting, when running the same sequential SCOTCH routine on different processors, to explore different solution spaces. The SCOTCH_randomProc routine allows the user to set an instance (processor) number that will be used to parametrize the random seed, hence providing different pseudo-random sequences for each instance number. However, when any of the two aforementioned compilation flags have been set, these sequence will still be deterministic: two runs of a sequence of LIBSCOTCH routines taking place after a call to SCOTCH_randomReset will always yield the same results.

In the case where the user wants to run concurrently LIBSCOTCH routines on different threads or sets of threads, determinism cannot be ensured using the global pseudo-random generator, because of the non-determinism in the way concurrent routines retrieve the values of the pseudo-random sequence. Moreover, because the global pseudo-number generator is not protected against race conditions, calling it concurrently from several threads may yield unpredictable results. Hence, in this case, users should use a different SCOTCH_Context for each master thread, that will contain its own pseudo-random generator (see Section 7.18).

7.17.1 SCOTCH_randomProc

Synopsis

```
void SCOTCH_randomProc (SCOTCH_Num  procnum)
scotchfrandomproc (integer*num  procnum)
```

Description

The SCOTCH_randomProc routine sets to procnum the internal instance of the LIBSCOTCH library. This instance number influences the random seed that is used to initialize pseudo-random number generators.

In order for this instance number to be taken into account as a seed for the global pseudo-random generator of the LIBSCOTCH library, SCOTCH_randomProc must be either called before any other library routine, or followed by a call to SCOTCH_randomReset. Subsequent calls to SCOTCH_randomReset will make use of this number as well.

The current value of procnum is copied along with the random seed, when the global pseudo-random number generator is cloned into a context by routine SCOTCH_contextRandomClone.

7.17.2 SCOTCH_randomReset

Synopsis

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

Description

The SCOTCH_randomReset routine resets the seed of the global pseudo-random generator used by default by the routines of the libScotch library.

Two consecutive calls to the same libScotch partitioning or ordering routines within the same program, separated by a call to SCOTCH_randomReset, will always yield the same results. Moreover, when Scotch has been compiled with any of the flags COMMON_RANDOM_FIXED_SEED or SCOTCH_DETERMINISTIC set, any two runs of the same program at different times will yield the same result.

7.17.3 SCOTCH_randomSeed

Synopsis

```
void SCOTCH_randomSeed (SCOTCH_Num  seedval)
scotchfrandomseed (integer*num  seedval)
```

Description

The SCOTCH_randomSeed routine sets to seedval the seed of the global pseudo-random generator used by default by some Scotch algorithms. All subsequent calls to SCOTCH_randomReset will use this value to reset the pseudo-random generator. In he case when no random seed is defined by the user, then depending whether Scotch has been compiled with any of the flags COMMON_RANDOM_FIXED_SEED or SCOTCH_DETERMINISTIC set or not, either the same pseudo-random seed will be always used, or a situation-dependent seed will be used, respectively.

7.17.4 SCOTCH_randomVal

Synopsis

```
SCOTCH_Num SCOTCH_randomVal (SCOTCH_Num  randmax)
scotchfrandomval (integer*num  randmax,
                  integer*num  randval)
```

Description

The SCOTCH_randomVal routine returns a positive integer random value from the global pseudo-random generator, in the range [0; textscrandmax[.

7.18 Context handling routines

By default, Scotch uses a global pseudo-random number generator and takes advantage of as many threads as it can discover on the system. This behavior makes

sense when only one call to the libScotch is made at a time. However, cases may arise where the user wants to perform different tasks on the same graph or mesh at the same time, by calling concurrently libScotch routines from different sets of threads. This is when SCOTCH_Context objects are necessary.

A SCOTCH_Context is a data structure that encapsulates an environment execution for the routines of the libScotch library. Essentially, it contains the set of threads that will be used for performing computations, and a private, independent pseudo-random number generator.

For a detailed description of the use of SCOTCH_Context's, please refer to the Scotch *User's Guide* [35].

7.18.1 SCOTCH_contextBindDgraph

Synopsis

```
int SCOTCH_contextBindDgraph (SCOTCH_Context *  contptr,
                              SCOTCH_Graph *    orggrafptr,
                              SCOTCH_Graph *    cntgrafptr)

scotchfcontextbinddgraph (doubleprecision (*)  contdat,
                          doubleprecision (*)  orggrafdat,
                          doubleprecision (*)  cntgrafdat,
                          integer              ierr)
```

Description

The SCOTCH_contextBindDgraph function initializes a context container object cntgraf, with a type compatible with a SCOTCH_Dgraph structure, to make it reference both the given genuine SCOTCH_Dgraph structure orggraf and the SCOTCH_Context structure cont. The context container can then be used by routines of the libPTScotch library that expect a SCOTCH_Dgraph, which will take advantage of all the features offered by the given context.

When the context container is no longer of use, call function SCOTCH_dgraph Exit to free its internal structures. The original distributed graph and the context can then also be freed by using the adequate routines.

Return values

SCOTCH_contextBindDgraph returns 0 if the context container graph structure has been successfully initialized, and 1 else.

7.19 Memory management

7.19.1 SCOTCH_memCur

Synopsis

```
SCOTCH_Idx SCOTCH_memCur (void)

scotchfmemcur (integer*idx  memcur)
```

Description

When SCOTCH is compiled with the COMMON_MEMORY_TRACE flag set, the SCOTCH_memCur routine returns the amount of memory, in bytes, that is currently allocated by SCOTCH on the current processing element, either by itself or on the behalf of the user. Else, the routine returns -1.

The returned figure does not account for the memory that has been allocated by the user and made visible to SCOTCH by means of routines such as SCOTCH_dgraphBuild calls. This memory is not under the control of SCOTCH, and it is the user's responsibility to free it after calling the relevant SCOTCH_*Exit routines.

Some third-party software used by SCOTCH, such as the strategy string parser, may allocate some memory for internal use and never free it. Consequently, there may be small discrepancies between memory occupation figures returned by SCOTCH and those returned by third-party tools. However, these discrepancies should not exceed a few kilobytes.

While memory occupation is internally recorded in a variable of type intptr_t, it is output as a SCOTCH_Idx for the sake of interface homogeneity, especially for Fortran. It is therefore the installer's responsibility to make sure that the support integer type of SCOTCH_Idx is large enough to not overflow. See Section 7.2.4 for more information.

7.19.2 SCOTCH_memMax

Synopsis

```
SCOTCH_Idx SCOTCH_memMax (void)
scotchfmemmax (integer*idx  memcur)
```

Description

When SCOTCH is compiled with the COMMON_MEMORY_TRACE flag set, the SCOTCH_memMax routine returns the maximum amount of memory, in bytes, ever allocated by SCOTCH on the current processing element, either by itself or on the behalf of the user. Else, the routine returns -1.

The returned figure does not account for the memory that has been allocated by the user and made visible to SCOTCH by means of routines such as SCOTCH_dgraphBuild calls. This memory is not under the control of SCOTCH, and it is the user's responsibility to free it after calling the relevant SCOTCH_*Exit routines.

Some third-party software used by SCOTCH, such as the strategy string parser, may allocate some memory for internal use and never free it. Consequently, there may be small discrepancies between memory occupation figures returned by SCOTCH and those returned by third-party tools. However, these discrepancies should not exceed a few kilobytes.

While memory occupation is internally recorded in a variable of type intptr_t, it is output as a SCOTCH_Idx for the sake of interface homogeneity, especially

for Fortran. It is therefore the installer's responsibility to make sure that the support integer type of `SCOTCH_Idx` is large enough to not overflow. See Section 7.2.4 for more information.

7.20 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 `libptscotchparmetis` 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 "`-lptscotchparmetis`" 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 9 for an example.

7.20.1 ParMETIS_V3_Mesh2Dual

Synopsis

```
void ParMETIS_V3_Mesh2Dual (const SCOTCH_Num * const  elmdist,
                            const SCOTCH_Num * const  eptr,
                            const SCOTCH_Num * const  eind,
                            const SCOTCH_Num * const  numflag,
                            const SCOTCH_Num * const  ncommonnodes,
                            SCOTCH_Num ** const       xadj,
                            SCOTCH_Num ** const       adjncy
                            MPI_Comm *                comm)

parmetis_v3_mesh2dual (integer*num (*)   elmdist,
                       integer*num (*)   eptr,
                       integer*num (*)   eind,
                       integer*num       numflag,
                       integer*num       ncommonnodes,
                       integer*num (**)  xadj,
                       integer*num (**)  adjncy,
                       integer           comm)
```

Description

The `ParMETIS_V3_Mesh2Dual` function computes in parallel the distributed element graph, also called dual mesh graph, from the given element mesh adjacency data. The input mesh is given through the three arrays `elmdist`, `eptr` and `eind`. The arrays which describe the adjacency of the dual graph are returned in the pointers pointed to by `xadj` and `adjncy`. It is the user's responsibility to free these two arrays after use, by way of plain `free` calls.

7.20.2 ParMETIS_V3_NodeND

Synopsis

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

parmetis_v3_nodend (integer*num (*)  vtxdist,
                    integer*num (*)  xadj,
                    integer*num (*)  adjncy,
                    integer*num      numflag,
                    integer*num (*)  options,
                    integer*num (*)  order,
                    integer*num (*)  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.

7.20.3 ParMETIS_V3_PartGeomKway

Synopsis

```
void ParMETIS_V3_PartGeomKway (const SCOTCH_Num * const  vtxdist,
                               const SCOTCH_Num * const  xadj,
                               const SCOTCH_Num * const  adjncy,
                               const SCOTCH_Num * const  vwgt,
                               const SCOTCH_Num * const  adjwgt,
                               const SCOTCH_Num * const  wgtflag,
                               const SCOTCH_Num * const  numflag,
                               const SCOTCH_Num * const  ndims,
                               const float * const       xyz,
                               const SCOTCH_Num * const  ncon,
                               const SCOTCH_Num * const  nparts,
                               const float * const       tpwgts,
                               const float * const       ubvec,
                               const SCOTCH_Num * const  options,
                               SCOTCH_Num * const        edgecut,
                               SCOTCH_Num * const        part,
                               MPI_Comm *                comm)

parmetis_v3_partgeomkway (integer*num (*)  vtxdist,
                          integer*num (*)  xadj,
                          integer*num (*)  adjncy,
                          integer*num (*)  vwgt,
                          integer*num (*)  adjwgt,
                          integer*num      wgtflag,
                          integer*num      numflag,
                          integer*num      ndims,
                          float (*)        xyz,
                          integer*num      ncon,
                          integer*num      nparts,
                          float (*)        tpwgts,
                          float (*)        ubvec,
                          integer*num (*)  options,
                          integer*num      edgecut,
                          integer*num (*)  part,
                          integer          comm)
```

Description

The `ParMETIS_V3_PartGeomKway` function computes a partition into `nparts` parts of the distributed graph passed as arrays `vtxdist`, `xadj` and `adjncy`, using the default PT-Scotch mapping strategy. The partition is returned in the form of the distributed vector `part`, which holds the indices of the parts to which every vertex belongs, from 0 to (`nparts` − 1).

Since Scotch does not handle geometry, the `ndims` and `xyz` arrays are not used, and this routine directly calls the `ParMETIS_V3_PartKway` stub.

7.20.4 `ParMETIS_V3_PartKway`

Synopsis

```
void ParMETIS_V3_PartKway (const SCOTCH_Num * const  vtxdist,
                           const SCOTCH_Num * const  xadj,
                           const SCOTCH_Num * const  adjncy,
                           const SCOTCH_Num * const  vwgt,
                           const SCOTCH_Num * const  adjwgt,
                           const SCOTCH_Num * const  wgtflag,
                           const SCOTCH_Num * const  numflag,
                           const SCOTCH_Num * const  ncon,
                           const SCOTCH_Num * const  nparts,
                           const float * const       tpwgts,
                           const float * const       ubvec,
                           const SCOTCH_Num * const  options,
                           SCOTCH_Num * const        edgecut,
                           SCOTCH_Num * const        part,
                           MPI_Comm *                comm)

parmetis_v3_partkway (integer*num (*)  vtxdist,
                      integer*num (*)  xadj,
                      integer*num (*)  adjncy,
                      integer*num (*)  vwgt,
                      integer*num (*)  adjwgt,
                      integer*num      wgtflag,
                      integer*num      numflag,
                      integer*num      ncon,
                      integer*num      nparts,
                      float (*)        tpwgts,
                      float (*)        ubvec,
                      integer*num (*)  options,
                      integer*num      edgecut,
                      integer*num (*)  part,
                      integer          comm)
```

Description

The `ParMETIS_V3_PartKway` function computes a partition into `nparts` parts of the distributed graph passed as arrays `vtxdist`, `xadj` and `adjncy`, using the default PT-Scotch mapping strategy. The partition is returned in the form of the distributed vector `part`, which holds the indices of the parts to which every vertex belongs, from 0 to (`nparts` − 1).

Since Scotch does not handle multiple constraints, only the first constraint is taken into account to define the respective weights of the parts. Consequently, only the first `nparts` cells of the `tpwgts` array are considered. The `ncon`, `ubvec` and `options` parameters are not used.

8 Installation

Version 7.0 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 GNU LGPL license. Therefore, it is not distributed as a set of binaries, but instead in the form of a source distribution,

which can be downloaded from the Scotch Inria GitLab repository at `https://gitlab.inria.fr/scotch/scotch` .

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

The extraction process will create a `scotch_7.0.18` 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.

8.1 Thread issues

To enable the use of POSIX threads in some routines, the `SCOTCH_PTHREAD` flag must be set. If this flag is defined, make sure to use the `MPI_Init_thread()` routine to initialize the communication subsystem. The `SCOTCH_PTHREAD_MPI` flag can also be set if the application software is likely to use the `MPI_THREAD_MULTIPLE` level (see Section 7.1).

8.2 File compression issues

To enable on-the-fly compression and decompression of various formats, the relevant flags must be defined. These flags are `COMMON_FILE_COMPRESS_BZ2` for `bzip2` (de)compression, `COMMON_FILE_COMPRESS_GZ` for `gzip` (de)compression, and `COMMON_FILE_COMPRESS_LZMA` for `lzma` decompression. Note that the corresponding development libraries must be installed on your system before compile time, and that compressed file handling can take place only on systems which support multi-threading or multi-processing. In the first case, you must set the `COMMON_PTHREAD` and `COMMON_PTHREAD_FILE` flags in order to take advantage of these features.

On Linux systems, the development libraries to install are `libbzip2_1-devel` for the `bzip2` format, `zlib1-devel` for the `gzip` format, and `liblzma0-devel` for the `lzma` format. The names of the libraries may vary according to operating systems and library versions. Ask your system engineer in case of trouble.

8.3 Machine word size issues

The integer values handled by Scotch are based on the `SCOTCH_Num` type, which equates by default to the `int` C type, corresponding to the `INTEGER` Fortran type, both of which being of machine word size. To coerce the length of the `SCOTCH_Num` integer type to 32 or 64 bits, one can use the "`-DINTSIZE32`" or "`-DINTSIZE64`" flags, respectively, or else use the "`-DINT=`" definition, at compile time. For instance, adding "`-DINT=long`" to the `CFLAGS` variable in the `Makefile.inc` file to be placed at the root of the source tree will make all `SCOTCH_Num` integers become `long` C integers.

Whenever doing so, make sure to use integer types of equivalent length to declare variables passed to Scotch routines from caller C and Fortran procedures. Also, because of API conflicts, the MeTiS compatibility library will not be usable. It is usually safer and cleaner to tune your C and Fortran compilers to make them interpret `int` and `INTEGER` types as 32 or 64 bit values, than to use the aforementioned flags and coerce type lengths in your own code.

Fortran users also have to take care of another size issue: since there are no pointers in Fortran 77, the Fortran interface of some routines converts pointers to be returned into integer indices with respect to a given array (*e.g.*, see Section 7.5.8). For 32_64 architectures, such indices can be larger than the size of a regular `INTEGER`. This is why the indices to be returned are defined by means of a specific integer type, `SCOTCH_Idx`. To coerce the length of this index type to 32 or 64 bits, one can use the "`-DIDXSIZE32`" or "`-DIDXSIZE64`" flags, respectively, or else use the "`-DIDX=`" definition, at compile time. For instance, adding "`-DIDX="long long"`" to the `CFLAGS` variable in the `Makefile.inc` file to be placed at the root of the source tree will equate all `SCOTCH_Idx` integers to C `long long` integers. By default, when the size of `SCOTCH_Idx` is not explicitly defined, it is assumed to be the same as the size of `SCOTCH_Num`.

9 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
  ```

- Compute on 4 processors an ordering of the compressed graph `brol.grf.gz`, and output the resulting ordering on compressed form.

  ```
  % mpirun -np 4 dgord brol.grf.gz brol.ord.gz
  ```

- 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 multilevel algorithm of PT-Scotch. He also studied parallel genetic refinement algorithms. Many thanks to him for the great job!
- Amaury Jacques improved the development environment and contributed to the consistency checking and non-regression testing routines;
- Selmane Lebdaoui coded the prototype of the threaded version of the distributed coarse graph building algorithm;
- Yves Secretan contributed to the MinGW32 port.

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. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, jan 2008. `http://www.labri.fr/~pelegrin/papers/scotch_parallelordering_parcomp.pdf`.

[8] F. Ercal, J. Ramanujam, and P. Sadayappan. Task allocation onto a hypercube by recursive mincut bipartitionning. *Journal of Parallel and Distributed Computing*, 10:35–44, 1990.

[9] 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.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractablility: A Guide to the Theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] S. W. Hammond. *Mapping unstructured grid computations to massively parallel computers*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New-York, February 1992.

[17] B. Hendrickson and R. Leland. Multidimensional spectral load balancing. Technical Report SAND93–0074, Sandia National Laboratories, January 1993.

[18] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93–1301, Sandia National Laboratories, June 1993.

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

[20] B. Hendrickson and R. Leland. The CHACO user's guide – version 2.0. Technical Report SAND94–2692, Sandia National Laboratories, 1994.

[21] B. Hendrickson and R. Leland. An empirical study of static load balancing algorithms. In *Proc. SHPCC'94, Knoxville*, pages 682–685. IEEE, May 1994.

[22] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. ACM/IEEE conference on Supercomputing (CDROM)*, dec 1995.

[23] B. Hendrickson, R. Leland, and R. Van Driessche. Skewed graph partitioning. In *Proceedings of the 8^{th} SIAM Conference on Parallel Processing for Scientific Computing*. IEEE, March 1997.

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

[25] 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.

[26] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. Technical Report 95-064, University of Minnesota, August 1995.

[27] 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.

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

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

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

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

[32] 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`.

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

[34] 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.

[35] F. Pellegrini. Scotch 5.1 User's Guide. Technical report, LaBRI, Université Bordeaux I, August 2008. Available from `http://www.labri.fr/~pelegrin/scotch/`.

[36] F. Pellegrini and J. Roman. Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. Research Report, LaBRI, Université Bordeaux I, August 1996. Available from `http://www.labri.fr/~pelegrin/papers/scotch_expanalysis.ps.gz`.

[37] 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.

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

[39] 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.

[40] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis*, 11(3):430–452, July 1990.

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

[42] 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.

[43] C. Walshaw, M. Cross, M. G. Everett, S. Johnson, and K. McManus. Partitioning & mapping of unstructured meshes to parallel machine topologies. In *Proc. Irregular'95*, LNCS 980, pages 121–126, 1995.