

Scotch and libScotch 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
francois.pellegrini@labri.fr

October 31, 2026

Abstract

This document describes the capabilities and operations of Scotch and libScotch, a software package and a software library devoted to static mapping, edge- and vertex-based graph partitioning, and sparse matrix block ordering of graphs and meshes/hypergraphs. It gives brief descriptions of the algorithms, details the input/output formats, instructions for use, installation procedures, and provides a number of examples.

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 **7**
- 1.1 Static mapping . . . 7
- 1.2 Sparse matrix ordering . . . 8
- 1.3 Contents of this document . . . 8

2 The Scotch project **9**
- 2.1 Description . . . 9
- 2.2 Availability . . . 9

3 Static mapping algorithms **10**
- 3.1 Cost function and performance criteria . . . 10
- 3.2 The Dual Recursive Bipartitioning algorithm . . . 11
  - 3.2.1 Partial cost function . . . 12
  - 3.2.2 Execution scheme . . . 13
  - 3.2.3 Clustering by mapping onto variable-sized architectures . . . 14
- 3.3 Static mapping methods . . . 14
- 3.4 Graph bipartitioning methods . . . 16

4 Sparse matrix ordering algorithms **18**
- 4.1 Performance criteria . . . 18
- 4.2 Minimum Degree . . . 18
- 4.3 Nested dissection . . . 18
- 4.4 Hybridization . . . 19
- 4.5 Ordering methods . . . 19
- 4.6 Graph separation methods . . . 20

5 Updates **21**
- 5.1 Changes in version 7.0 from version 6.1 . . . 21
- 5.2 Changes in version 6.1 from version 6.0 . . . 22
- 5.3 Changes in version 6.0 from version 5.1 . . . 22
- 5.4 Changes in version 5.1 from version 5.0 . . . 23

6 Files and data structures **23**
- 6.1 Graph files . . . 23
- 6.2 Mesh files . . . 24
- 6.3 Geometry files . . . 26
- 6.4 Target files . . . 27
  - 6.4.1 Decomposition-defined architecture files . . . 27
  - 6.4.2 Algorithmically-coded architecture files . . . 28
  - 6.4.3 Variable-sized architecture files . . . 31
- 6.5 Mapping files . . . 31
- 6.6 Ordering files . . . 32
- 6.7 Vertex list files . . . 33

7 Programs **33**
- 7.1 Invocation . . . 33
- 7.2 Using multi-threading . . . 35
- 7.3 Using compressed files . . . 35
- 7.4 Description . . . 36
  - 7.4.1 `acpl` . . . 36

7.4.2 amk_* . . . 36
7.4.3 amk_grf . . . 38
7.4.4 atst . . . 39
7.4.5 gcv . . . 39
7.4.6 gmap / gpart . . . 40
7.4.7 gmk_* . . . 43
7.4.8 gmk_msh . . . 44
7.4.9 gmtst . . . 44
7.4.10 gord . . . 45
7.4.11 gotst . . . 47
7.4.12 gout . . . 47
7.4.13 gtst . . . 50
7.4.14 mcv . . . 51
7.4.15 mmk_* . . . 51
7.4.16 mord . . . 52
7.4.17 mtst . . . 53

8 Library **54**
8.1 Calling the routines of libScotch . . . 54
8.1.1 Calling from C . . . 54
8.1.2 Calling from Fortran . . . 55
8.1.3 Compiling and linking . . . 56
8.1.4 Dynamic library issues . . . 56
8.1.5 Machine word size issues . . . 57
8.1.6 Using multi-threading . . . 57
8.2 Data types . . . 58
8.2.1 SCOTCH_Arch architecture type . . . 58
8.2.2 SCOTCH_Graph graph type . . . 58
8.2.3 SCOTCH_Mesh mesh type . . . 61
8.2.4 SCOTCH_Geom geometry type . . . 62
8.2.5 SCOTCH_Ordering block ordering format . . . 64
8.3 Strategy strings . . . 65
8.3.1 Using default strategy strings . . . 65
8.3.2 Mapping strategy strings . . . 66
8.3.3 Graph bipartitioning strategy strings . . . 69
8.3.4 Vertex partitioning (with overlap) strategy strings . . . 73
8.3.5 Ordering strategy strings . . . 75
8.3.6 Node separation strategy strings . . . 78
8.4 Target architecture handling routines . . . 82
8.4.1 SCOTCH_archAlloc . . . 82
8.4.2 SCOTCH_archExit . . . 82
8.4.3 SCOTCH_archInit . . . 83
8.4.4 SCOTCH_archLoad . . . 83
8.4.5 SCOTCH_archName . . . 83
8.4.6 SCOTCH_archSave . . . 84
8.4.7 SCOTCH_archSize . . . 84
8.4.8 SCOTCH_archSizeof . . . 85
8.5 Target architecture creation routines . . . 85
8.5.1 SCOTCH_archBuild0 / SCOTCH_archBuild . . . 85
8.5.2 SCOTCH_archBuild2 . . . 86
8.5.3 SCOTCH_archCmplt . . . 87

8.5.4 SCOTCH_archCmpltw 87
8.5.5 SCOTCH_archHcub 88
8.5.6 SCOTCH_archLtleaf 88
8.5.7 SCOTCH_archMesh2 89
8.5.8 SCOTCH_archMesh3 90
8.5.9 SCOTCH_archMeshX 90
8.5.10 SCOTCH_archSub 91
8.5.11 SCOTCH_archTleaf 91
8.5.12 SCOTCH_archTorus2 92
8.5.13 SCOTCH_archTorus3 92
8.5.14 SCOTCH_archTorusX 93
8.5.15 SCOTCH_archVcmplt 93
8.5.16 SCOTCH_archVhcub 94
8.6 Target domain handling routines 94
8.6.1 SCOTCH_archDomAlloc 94
8.6.2 SCOTCH_archDomBipart 95
8.6.3 SCOTCH_archDomFrst 95
8.6.4 SCOTCH_archDomSize 96
8.6.5 SCOTCH_archDomSizeof 96
8.6.6 SCOTCH_archDomTerm 96
8.6.7 SCOTCH_archDomWght 97
8.6.8 SCOTCH_archDomDist 97
8.6.9 SCOTCH_archDomNum 98
8.7 Graph handling routines 98
8.7.1 SCOTCH_graphAlloc 98
8.7.2 SCOTCH_graphBase 99
8.7.3 SCOTCH_graphBuild 99
8.7.4 SCOTCH_graphCheck 101
8.7.5 SCOTCH_graphCoarsen 101
8.7.6 SCOTCH_graphCoarsenBuild 102
8.7.7 SCOTCH_graphCoarsenMatch 103
8.7.8 SCOTCH_graphColor 104
8.7.9 SCOTCH_graphData 105
8.7.10 SCOTCH_graphDiamPV 106
8.7.11 SCOTCH_graphDump 107
8.7.12 SCOTCH_graphExit 107
8.7.13 SCOTCH_graphFree 107
8.7.14 SCOTCH_graphInduceList 108
8.7.15 SCOTCH_graphInducePart 108
8.7.16 SCOTCH_graphInit 109
8.7.17 SCOTCH_graphLoad 109
8.7.18 SCOTCH_graphSave 110
8.7.19 SCOTCH_graphSize 111
8.7.20 SCOTCH_graphSizeof 111
8.7.21 SCOTCH_graphStat 111
8.8 High-level graph partitioning, mapping and clustering routines . . . 112
8.8.1 SCOTCH_graphMap 113
8.8.2 SCOTCH_graphMapFixed 114
8.8.3 SCOTCH_graphPart 114
8.8.4 SCOTCH_graphPartFixed 115
8.8.5 SCOTCH_graphPartOvl 116

8.8.6 SCOTCH_graphPartOvlView . . . 117
8.8.7 SCOTCH_graphRemap . . . 117
8.8.8 SCOTCH_graphRemapFixed . . . 119
8.8.9 SCOTCH_graphRepart . . . 120
8.8.10 SCOTCH_graphRepartFixed . . . 121
8.9 Low-level graph partitioning, mapping and clustering routines . . . 122
8.9.1 SCOTCH_graphMapCompute . . . 122
8.9.2 SCOTCH_graphMapExit . . . 123
8.9.3 SCOTCH_graphMapFixedCompute . . . 123
8.9.4 SCOTCH_graphMapInit . . . 124
8.9.5 SCOTCH_graphMapLoad . . . 124
8.9.6 SCOTCH_graphMapSave . . . 125
8.9.7 SCOTCH_graphMapView . . . 126
8.9.8 SCOTCH_graphRemapCompute . . . 126
8.9.9 SCOTCH_graphRemapFixedCompute . . . 127
8.9.10 SCOTCH_graphTabLoad . . . 128
8.9.11 SCOTCH_graphTabSave . . . 129
8.10 High-level graph ordering routines . . . 130
8.10.1 SCOTCH_graphOrder . . . 130
8.11 Low-level graph ordering routines . . . 131
8.11.1 SCOTCH_graphOrderCheck . . . 131
8.11.2 SCOTCH_graphOrderCompute . . . 131
8.11.3 SCOTCH_graphOrderComputeList . . . 132
8.11.4 SCOTCH_graphOrderExit . . . 133
8.11.5 SCOTCH_graphOrderInit . . . 133
8.11.6 SCOTCH_graphOrderLoad . . . 134
8.11.7 SCOTCH_graphOrderSave . . . 135
8.11.8 SCOTCH_graphOrderSaveMap . . . 135
8.11.9 SCOTCH_graphOrderSaveTree . . . 136
8.12 Mesh handling routines . . . 137
8.12.1 SCOTCH_meshAlloc . . . 137
8.12.2 SCOTCH_meshBuild . . . 137
8.12.3 SCOTCH_meshBuildElem . . . 138
8.12.4 SCOTCH_meshCheck . . . 140
8.12.5 SCOTCH_meshData . . . 140
8.12.6 SCOTCH_meshExit . . . 142
8.12.7 SCOTCH_meshGraph . . . 142
8.12.8 SCOTCH_meshGraphDual . . . 143
8.12.9 SCOTCH_meshInit . . . 143
8.12.10 SCOTCH_meshLoad . . . 144
8.12.11 SCOTCH_meshSave . . . 144
8.12.12 SCOTCH_meshSize . . . 145
8.12.13 SCOTCH_meshSizeof . . . 146
8.12.14 SCOTCH_meshStat . . . 146
8.13 High-level mesh ordering routines . . . 147
8.13.1 SCOTCH_meshOrder . . . 147
8.14 Low-level mesh ordering routines . . . 148
8.14.1 SCOTCH_meshOrderCheck . . . 148
8.14.2 SCOTCH_meshOrderCompute . . . 149
8.14.3 SCOTCH_meshOrderExit . . . 149
8.14.4 SCOTCH_meshOrderInit . . . 150

8.14.5 SCOTCH_meshOrderSave 150
8.14.6 SCOTCH_meshOrderSaveMap 151
8.14.7 SCOTCH_meshOrderSaveTree 152
8.15 Strategy handling routines 152
8.15.1 SCOTCH_stratAlloc 152
8.15.2 SCOTCH_stratExit 153
8.15.3 SCOTCH_stratInit 153
8.15.4 SCOTCH_stratSave 153
8.15.5 SCOTCH_stratSizeof 154
8.16 Strategy creation routines 154
8.16.1 SCOTCH_stratGraphBipart 154
8.16.2 SCOTCH_stratGraphClusterBuild 155
8.16.3 SCOTCH_stratGraphMap 156
8.16.4 SCOTCH_stratGraphMapBuild 156
8.16.5 SCOTCH_stratGraphPartOvl 157
8.16.6 SCOTCH_stratGraphPartOvlBuild 157
8.16.7 SCOTCH_stratGraphOrder 158
8.16.8 SCOTCH_stratGraphOrderBuild 158
8.16.9 SCOTCH_stratMeshOrder 159
8.16.10 SCOTCH_stratMeshOrderBuild 159
8.17 Geometry handling routines 160
8.17.1 SCOTCH_geomAlloc 160
8.17.2 SCOTCH_geomData 161
8.17.3 SCOTCH_geomExit 162
8.17.4 SCOTCH_geomInit 162
8.17.5 SCOTCH_geomSizeof 162
8.17.6 SCOTCH_graphGeomLoadChac 163
8.17.7 SCOTCH_graphGeomLoadHabo 163
8.17.8 SCOTCH_graphGeomLoadScot 164
8.17.9 SCOTCH_graphGeomSaveChac 165
8.17.10 SCOTCH_graphGeomSaveScot 165
8.17.11 SCOTCH_meshGeomLoadHabo 166
8.17.12 SCOTCH_meshGeomLoadScot 166
8.17.13 SCOTCH_meshGeomSaveScot 167
8.18 Other data structure handling routines 168
8.18.1 SCOTCH_mapAlloc 168
8.18.2 SCOTCH_mapSizeof 168
8.18.3 SCOTCH_orderAlloc 168
8.18.4 SCOTCH_orderSizeof 169
8.19 Error handling routines 169
8.19.1 SCOTCH_errorPrint 169
8.19.2 SCOTCH_errorPrintW 170
8.19.3 SCOTCH_errorProg 170
8.20 Random generator handling 170
8.20.1 SCOTCH_randomProc 171
8.20.2 SCOTCH_randomReset 171
8.20.3 SCOTCH_randomSeed 172
8.20.4 SCOTCH_randomVal 172
8.21 Context handling routines 172
8.21.1 SCOTCH_contextInit 173
8.21.2 SCOTCH_contextExit 174

8.21.3 SCOTCH_contextOptionGetNum . . . 174
8.21.4 SCOTCH_contextOptionSetNum . . . 175
8.21.5 SCOTCH_contextRandomClone . . . 175
8.21.6 SCOTCH_contextRandomReset . . . 176
8.21.7 SCOTCH_contextRandomSeed . . . 176
8.21.8 SCOTCH_contextSizeof . . . 177
8.21.9 SCOTCH_contextThreadImport1 . . . 177
8.21.10 SCOTCH_contextThreadImport2 . . . 177
8.21.11 SCOTCH_contextThreadSpawn . . . 179
8.21.12 SCOTCH_contextBindGraph . . . 180
8.21.13 SCOTCH_contextBindMesh . . . 181
8.22 Memory management . . . 182
8.22.1 SCOTCH_memCur . . . 182
8.22.2 SCOTCH_memFree . . . 182
8.22.3 SCOTCH_memMax . . . 183
8.23 Miscellaneous routines . . . 183
8.23.1 SCOTCH_numSizeof . . . 183
8.23.2 SCOTCH_version . . . 184
8.24 MeTiS compatibility library . . . 184
8.24.1 METIS_EdgeND . . . 184
8.24.2 METIS_MeshToDual . . . 185
8.24.3 METIS_NodeND . . . 186
8.24.4 METIS_NodeWND . . . 186
8.24.5 METIS_PartGraphKway . . . 187
8.24.6 METIS_PartGraphRecursive . . . 188
8.24.7 METIS_PartGraphVKway . . . 189
8.24.8 METIS_PartMeshDual . . . 190
8.24.9 METIS_SetDefaultOptions . . . 191

9 Installation 192
9.1 Thread issues . . . 192
9.2 File compression issues . . . 192
9.3 Machine word size issues . . . 193

10 Examples 193

11 Adding new features to Scotch 195
11.1 Graphs and meshes . . . 195
11.2 Methods and partition data . . . 196
11.3 Adding a new method to Scotch . . . 196
11.4 Licensing of new methods and of derived works . . . 198

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 inter-communicating 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 [15]. 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 [12, 23]. 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 [5, 24, 33, 35, 52]. 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 [23, 60].

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 computation costs.

An efficient way to compute fill reducing orderings of symmetric sparse matrices is to use recursive nested dissection [19]. It amounts to computing a vertex set S that separates the graph into two parts A and B, ordering S with the highest indices that are still available, and proceeding 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.

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, in order to minimize fill-in and to increase concurrency in the factorization process.

1.3 Contents of this document

This document describes the capabilities and operations of Scotch, a software package devoted to static mapping, graph and mesh partitioning, and sparse matrix block ordering. Scotch allows the user to map efficiently any kind of weighted process graph onto any kind of weighted architecture graph, and provides high-quality block orderings of sparse matrices. The rest of this manual is organized

as follows. Section 2 presents the goals of the SCOTCH project. Sections 3 and 4 outline the most important aspects of the mapping and ordering algorithms that it implements, respectively. Section 5 summarizes the most important changes between version 5.0 and previous versions. Section 6 defines the formats of the files used in SCOTCH, section 7 describes the programs of the SCOTCH distribution, and section 8 defines the interface and operations of the LIBSCOTCH library. Section 9 explains how to obtain and install the SCOTCH distribution. Finally, some practical examples are given in section 10, and instructions on how to implement new methods in the LIBSCOTCH library are provided in section 11.

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 de Bordeaux and within the Tadaam team-project of INRIA Bordeaux Sud-Ouest. Its goal is to study the application of graph theory to scientific computing.

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 [43], all of which have been implemented in the SCOTCH software package [47]. 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 [48, 49]. The ordering capabilities of SCOTCH have then been extended to native mesh structures, thanks to hypergraph partitioning algorithms. Diffusion-based graph partitioning methods have also been added [9, 44].

Version 5.0 of SCOTCH was the first one to comprise parallel graph ordering routines. The parallel features of SCOTCH are referred to as PT-SCOTCH ("*Parallel Threaded* SCOTCH"). 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 parallel features of SCOTCH should refer to the PT-SCOTCH 7.0 *User's Guide* [45].

Version 6.0 of SCOTCH is oriented towards the development of new features, namely graph repartitioning and remapping [14]. A whole set of direct k-way graph partitioning and mapping algorithms has also been implemented. Also, new target architectures have been created, to allow SCOTCH to map efficiently onto parts of regular target architectures [50], as it is the case when considering a potentially non-connected partition of a big machine, as provided by a batch scheduler.

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 [7], which has basically the same features as the GNU LGPL ("*Lesser General Public License*"): 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.

Please refer to section 9 to see how to obtain the free/libre distribution of SCOTCH.

3 Static mapping algorithms

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 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 [12, 23, 27], 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 routing 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 [23] on the CM-2, and by Hendrickson and Leland [28] 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 inter-processor 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

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

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

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.

3.2 The Dual Recursive Bipartitioning algorithm

This mapping algorithm, which is the primary way to compute initial static mappings, uses a *divide and conquer* approach to recursively allocate subsets of processes to subsets of processors [43, 46]. 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 [46, 50]. 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.

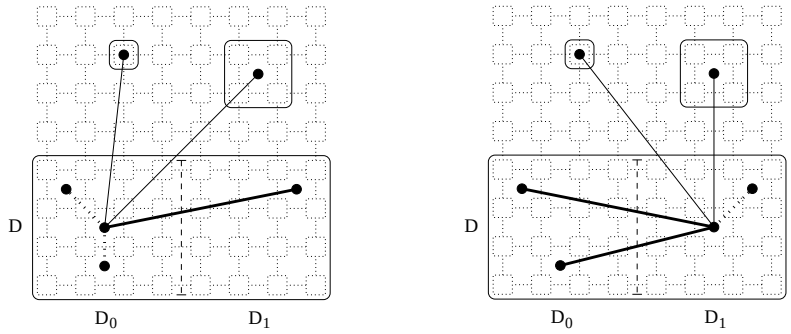
All these routines are seen as black boxes by the mapping program, which can thus accept any kind of target architecture and process bipartitioning functions.

3.2.1 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 *skewed graph partitioning* by some authors [29].



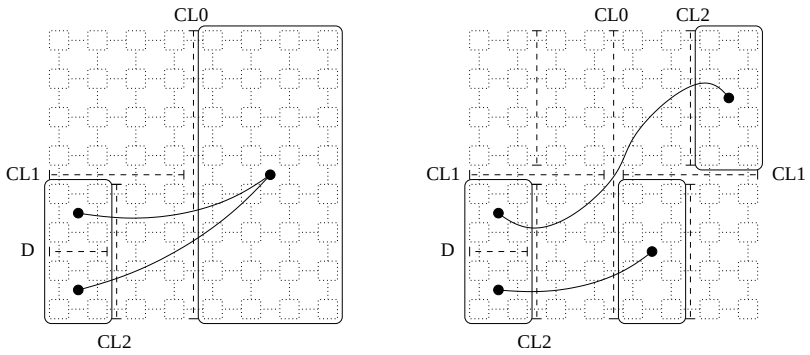
a. Initial position. **b.** After one vertex is moved.

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.

3.2.2 Execution scheme

From an algorithmic point of view, our mapper behaves as a greedy algorithm, since the mapping of a process to a subdomain is never reconsidered, and at each step of which iterative algorithms can be applied. The double recursive call performed at each step induces a recursion scheme in the shape of a binary tree, each vertex of which corresponds to a bipartitioning job, that is, the bipartitioning of both a domain and its associated subgraph.

In the case of depth-first sequencing, as written in the above sketch, bipartitioning jobs run in the left branches of the tree have no information on the distance between the vertices they handle and neighbor vertices to be processed in the right branches. On the contrary, sequencing the jobs according to a by-level (breadth-first) travel of the tree allows any bipartitioning job of a given level to have information on the subdomains to which all the processes have been assigned at the previous level. Thus, when deciding in which subdomain to put a given process, a bipartitioning job can account for the communication costs induced by its neighbor processes, whether they are handled by the job itself or not, since it can estimate in f'_C the dilation of the corresponding edges. This results in an interesting feedback effect: once an edge has been kept in a cut between two subdomains, the distance between its end vertices will be accounted for in the partial communication cost function to be minimized, and following jobs will be more likely to keep these vertices close to each other, as illustrated in Figure 2. The relative efficiency of depth-first and breadth-first sequencing schemes with respect to the structure of the source and target graphs is discussed in [46].



a. Depth-first sequencing. **b.** Breadth-first sequencing.

Figure 2: Influence of depth-first and breadth-first sequencings on the bipartitioning of a domain D belonging to the leftmost branch of the bipartitioning tree. With breadth-first sequencing, the partial mapping data regarding vertices belonging to the right branches of the bipartitioning tree are more accurate (C.L. stands for "Cut Level").

3.2.3 Clustering by 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 checks the validity of a criterion at each bipartitioning step (see for instance Section 8.16.2), and maps all of the subgraph vertices to one of the subdomains when it becomes false.

3.3 Static mapping methods

The core of our static mapping software uses graph mapping methods as black boxes. It maintains an internal image of the current mapping, which records the target vertex index onto which each of the source graph vertices is mapped. It is therefore possible to apply several mapping methods in sequence, such that the first method computes an initial mapping to be further refined by the following methods, thus enabling us to define *static mapping strategies*. The currently implemented static mapping methods are listed below.

Multilevel

This framework, which has been studied by several authors [5, 25, 33] 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 map by finding matchings that collapse vertices and edges, computes a mapping of the

coarsest graph obtained, and prolongs the result back to the original graph, as shown in Figure 3. The multilevel method, when used in conjunction with

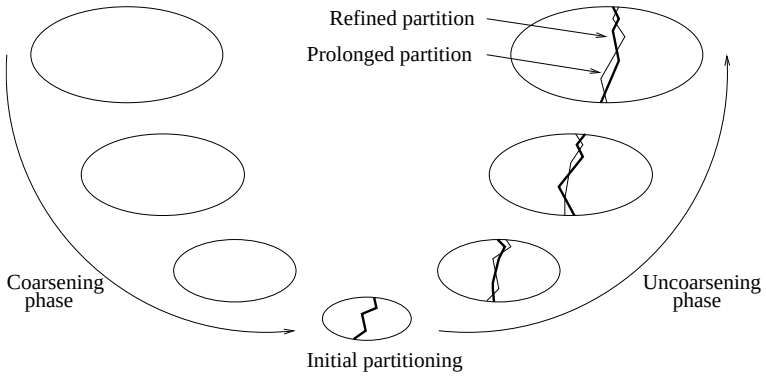


Figure 3: The multilevel partitioning process. In the uncoarsening phase, the light and bold lines represent for each level the prolonged partition obtained from the coarser graph, and the partition obtained after refinement, respectively.

some local optimization methods to refine the projected partitions at every level, usually leads to a significant improvement in quality with respect to methods operating only on the finest graph. By coarsening the graphs, the multilevel algorithm broadens the scope of local optimization algorithms: it makes possible for them to account for topological structures of the original graph that would else be of a too high level for them to be encompassed in their local optimization process.

Band

Like the multilevel method above, the band method is a framework, in the sense that it does not itself compute partitions, but rather helps other partitioning algorithms perform better. It is a refinement 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 frontiers of the parts), calls a partitioning strategy on this band graph, and projects 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 multilevel 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 [9].

Fiduccia-Mattheyses

This is a direct k-way version of the traditional Fiduccia-Mattheyses heuristics used for computing bipartitions, that will be presented in the next section. By default, boundary vertices can only be moved to parts to which at least one of their neighbors belong.

Diffusion

This is also a k-way version of an algorithm that has been first used in the context of bipartitioning, and which will be presented in the next section. The

k-way version differs from the latter as it diffuses k sorts of liquids rather than just two as in the bipartitioning case.

Exactifier
This greedy algorithm refines its input mapping so as to reduce load imbalance as much as possible. Since this method does not consider load balance minimization, its use should be restricted to cases where achieving load balance is critical and where recursive bipartitioning may fail to achieve it. It is especially the case when vertex loads are very irregular: some subdomains may receive only a few heavy vertices, yielding load balance artifacts when no light vertices are locally available to compensate.

Graph vertices are sorted by decreasing weights, and considered in turn. If the current vertex can fit in its initial part without causing imbalance by excess, it is added to it, and the algorithm goes on. Else, a candidate part is found by exploring other subdomains in an order based on an implicit recursive bipartitioning of the architecture graph. Consequently, such vertices will be placed in subdomains that tend to be as close as possible to the original location of the vertex. This method is most likely to result in disconnected parts.

Dual recursive bipartitioning
This algorithm implements the dual recursive bipartitioning algorithm that has been presented in Section 3.2. The DRB algorithms can be used either directly on the original graph to partition, or on the coarsest graph yielded by the direct k-way multilevel framework. It uses graph bipartitioning methods, described below, to compute its bipartitions.

3.4 Graph bipartitioning methods

The core of our dual recursive bipartitioning mapping algorithm uses process graph 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 *graph bipartitioning strategies*. The currently implemented graph bipartitioning methods are listed below.

Diffusion
This global optimization method, presented in [44], 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.

Exactifier

This greedy algorithm refines the current partition so as to reduce load imbalance as much as possible, while keeping the value of the communication cost function as small as possible. The vertex set is scanned in order of decreasing vertex weights, and vertices are moved from one subdomain to the other if doing so reduces load imbalance. When several vertices have same weight, the vertex whose swap decreases most the communication cost function is selected first. This method is used in post-processing of other methods when load balance is mandatory. For weighted graphs, the strict enforcement of load balance may cause the swapping of isolated vertices of small weight, thus greatly increasing the cut. Therefore, great care should be taken when using this method if connectivity or cut minimization are mandatory.

Fiduccia-Mattheyses

The Fiduccia-Mattheyses heuristics [13] is an almost-linear improvement of the famous Kernighan-Lin algorithm [37]. It tries to improve the bipartition that is input to it by incrementally moving vertices between the subsets of the partition, as long as it can find sequences of moves that lower its communication cost. By considering sequences of moves instead of single swaps, the algorithm allows hill-climbing from local minima of the cost function. As an extension to the original Fiduccia-Mattheyses algorithm, we have developed new data structures, based on logarithmic indexings of arrays, that allow us to handle weighted graphs while preserving the almost-linearity in time of the algorithm [46].

As several authors quoted before [26, 34], the Fiduccia-Mattheyses algorithm gives better results when trying to optimize a good starting partition. Therefore, it should not be used on its own, but rather after greedy starting methods such as the Gibbs-Poole-Stockmeyer or the greedy graph growing methods.

Gibbs-Poole-Stockmeyer

This greedy bipartitioning method derives from an algorithm proposed by Gibbs, Poole, and Stockmeyer to minimize the dilation of graph orderings, that is, the maximum absolute value of the difference between the numbers of neighbor vertices [20]. The graph is sliced by using a breadth-first spanning tree rooted at a randomly chosen vertex, and this process is iterated by selecting a new root vertex within the last layer as long as the number of layers increases. Then, starting from the current root vertex, vertices are assigned layer after layer to the first subdomain, until half of the total weight has been processed. Remaining vertices are then allocated to the second subdomain.

As for the original Gibbs, Poole, and Stockmeyer algorithm, it is assumed that the maximization of the number of layers results in the minimization of the sizes –and therefore of the cocycles– of the layers. This property has already been used by George and Liu to reorder sparse linear systems using the nested dissection method [19], and by Simon in [58].

Greedy graph growing

This greedy algorithm, which has been proposed by Karypis and Kumar [33], belongs to the GRASP ("*Greedy Randomized Adaptive Search Procedure*") class [39]. It consists in selecting an initial vertex at random, and repeatedly adding vertices to this growing subset, such that each added vertex results in the smallest increase in the communication cost function. This process, which stops when load balance is achieved, is repeated several times in order

to explore (mostly in a gradient-like fashion) different areas of the solution space, and the best partition found is kept.

Multilevel
This is a graph bipartition-oriented version of the static mapping multilevel method described in the previous section, page 14.

4 Sparse matrix ordering algorithms

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).

4.1 Performance criteria

The quality of orderings is evaluated with respect to several criteria. The first one, NNZ, is the number of non-zero terms in the factored reordered matrix. The second one, OPC, is the operation count, that is the number of arithmetic operations required to factor the matrix. The operation count that we have considered takes into consideration all operations (additions, subtractions, multiplications, divisions) required by Cholesky factorization, except square roots; it is equal to $\sum_c n_c^2$, where n_c is the number of non-zeros of column c of the factored matrix, diagonal included. A third criterion for quality is the shape of the elimination tree; concurrency in parallel solving is all the higher as the elimination tree is broad and short. To measure its quality, several parameters can be defined: $h_{\min}$, $h_{\max}$, and h_{avg} denote the minimum, maximum, and average heights of the tree[1], respectively, and h_{dlt} is the variance, expressed as a percentage of h_{avg}. Since small separators result in small chains in the elimination tree, h_{avg} should also indirectly reflect the quality of separators.

4.2 Minimum Degree

The minimum degree algorithm [59] is a local heuristic that performs its pivot selection by iteratively selecting from the graph a node of minimum degree.

The minimum degree algorithm 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, 18, 42]). However, the algorithm is intrinsically sequential, and very little can be theoretically proved about its efficiency.

4.3 Nested dissection

The nested dissection algorithm [19] is a global, heuristic, recursive 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.

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

Many theoretical results have been carried out on nested dissection ordering [8, 41], 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. Most often, vertex separators are computed by using direct heuristics [30, 40], or from edge separators [51, and included references] by minimum cover techniques [10, 32], but other techniques such as spectral vertex partitioning have also been used [52].

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 [21, 33, 48] to the ones obtained with the minimum degree algorithm [42]. 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 [3, 16, 17, 21, 48, 56, and included references].

4.4 Hybridization

Due to their complementary nature, several schemes have been proposed to hybridize the two methods [30, 36, 48]. However, to our knowledge, only loose couplings have been achieved: incomplete nested dissection is performed on the graph to order, and the resulting subgraphs are passed to some minimum degree algorithm. This results in the fact that the minimum degree algorithm does not have exact degree values for all of the boundary vertices of the subgraphs, leading to a misbehavior of the vertex selection process.

Our ordering program implements 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. First, the nested dissection algorithm provides exact degree values for the boundary vertices of the subgraphs passed to the minimum degree algorithm (called *halo* minimum degree since it has a partial visibility of the neighborhood of the subgraph). Second, the minimum degree algorithm returns the assembly tree that it computes for each subgraph, thus allowing for supervariable amalgamation, in order to obtain column-blocks of a size suitable for BLAS3 block computations.

As for our mapping program, it is possible to combine ordering methods into ordering strategies, which allow the user to select the proper methods with respect to the characteristics of the subgraphs.

The ordering program is completely parametrized by its ordering strategy. The nested dissection method allows the user to take advantage of all of the graph partitioning routines that have been developed in the earlier stages of the Scotch project. Internal ordering strategies for the separators are relevant in the case of sequential or parallel [22, 53, 54, 55] block solving, to select ordering algorithms that minimize the number of extra-diagonal blocks [8], thus allowing for efficient use of BLAS3 primitives, and to reduce inter-processor communication.

4.5 Ordering methods

The core of our ordering algorithm uses graph ordering methods as black boxes, which allows the orderer to run any type of ordering method. In addition to yielding orderings of the subgraphs that are passed to them, these methods may compute column block partitions of the subgraphs, that are recorded in a separate tree structure. The currently implemented graph ordering methods are listed below.

Halo approximate minimum degree

The halo approximate minimum degree method [49] is an improvement of the approximate minimum degree [1] algorithm, suited for use on subgraphs produced by nested dissection methods. Its interest compared to classical minimum degree algorithms is that boundary vertices are processed using their real degree in the global graph rather than their (much smaller) degree in the subgraph, resulting in smaller fill-in and operation count. This method also implements amalgamation techniques that result in efficient block computations in the factoring and the solving processes.

Halo approximate minimum fill

The halo approximate minimum fill method is a variant of the halo approximate minimum degree algorithm, where the criterion to select the next vertex to permute is not based on its current estimated degree but on the minimization of the induced fill.

Graph compression

The graph compression method [2] merges cliques of vertices into single nodes, so as to speed-up the ordering of the compressed graph. It also results in some improvement of the quality of separators, especially for stiffness matrices.

Gibbs-Poole-Stockmeyer

This method is mainly used on separators to reduce the number and extent of extra-diagonal blocks.

Simple method

Vertices are ordered consecutively, in the same order as they are stored in the graph. This is the fastest method to use on separators when the shape of extra-diagonal structures is not a concern.

Nested dissection

Incomplete nested dissection method. Separators are computed recursively on subgraphs, and specific ordering methods are applied to the separators and to the resulting subgraphs (see sections 4.3 and 4.4).

Disconnected subgraph detection

This method may be used as a pre-processing step so as to apply the same ordering strategy on each of the disconnected components of a graph. While finding the connected components of a graph is expensive, it may bring an improvement on graph ordering quality in some cases.

4.6 Graph separation methods

The core of our incomplete nested dissection algorithm uses graph separation methods as black boxes. It allows the orderer to run any type of graph separation method compatible with our criteria for quality, that is, reducing the size of the vertex separator while maintaining the loads of the separated parts within some user-specified tolerance. Separation jobs maintain an internal image of the current vertex separator, indicating for every vertex of the job whether it is currently assigned to one of the two parts, or to the separator. 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 the definition of separation strategies.

The currently implemented graph separation methods are listed below.

Fiduccia-Mattheyses
This is a vertex-oriented version of the original, edge-oriented, Fiduccia-Mattheyses heuristics described in page 17.

Greedy graph growing
This is a vertex-oriented version of the edge-oriented greedy graph growing algorithm described in page 17.

Multilevel
This is a vertex-oriented version of the edge-oriented multilevel algorithm described in page 14.

Thinner
This greedy algorithm refines the current separator by removing all of the exceeding vertices, that is, vertices that do not have neighbors in both parts. It is provided as a simple gradient refinement algorithm for the multilevel method, and is clearly outperformed by the Fiduccia-Mattheyses algorithm.

Vertex cover
This algorithm computes a vertex separator by first computing an edge separator, that is, a bipartition of the graph, and then turning it into a vertex separator by using the method proposed by Pothen and Fang [51]. This method requires the computation of maximal matchings in the bipartite graphs associated with the edge cuts, which are built using Duff's variant [10] of the Hopcroft and Karp algorithm [32]. Edge separators are computed by using a bipartitioning strategy, which can use any of the graph bipartitioning methods described in section 3.4, page 16.

5 Updates

5.1 Changes in version 7.0 from version 6.1

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 libScotch library and, consequently, by the Scotch standalone programs that call them. Users can also control concurrency by using `SCOTCH_Context` objects. These objects define user-configurable execution contexts, in which libScotch library routines can be executed independently from others; see Section 8.21, page 172 for further information.

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 (see Section 8.21.4).

Support for CMake has been added.

A new, faster, pseudo-random number generator has been implemented.

The y coordinate of the grid geometry files provided by the `gmk_m*` programs has been set to the ascending order. Use option `-y` to restore the previous behavior

of descending order.

A multi-threaded genetic graph bipartitioning method has been added.

The `cmpltws` target architecture has been added.

5.2 Changes in version 6.1 from version 6.0

The k-way refinement routine of the module that computes graph partitions with overlap has been completely rewritten.

The halo minimum degree and halo minimum fill algorithms now take into account, for computing vertex degrees, the vertex weights attached to node vertices. These weights may represent the number of degrees of freedom associated with a vertex during subsequent matrix computations. They can result from a graph compression process, such as the one implemented in Scotch (see page 76).

The program `gout` can now output VTK files.

5.3 Changes in version 6.0 from version 5.1

The new `sub` abstract target architecture allows one to map a graph onto a subset of any given target architecture (including another `sub` architecture). This feature is meant to perform mappings onto potentially disconnected subsets of a parallel machine, e.g. the set of nodes assigned by a batch scheduler; see Section 8.5.10, page 91 for further information.

Also, in order to allow decomposition-defined architectures to scale-up to the sizes of modern machines, a new version of the `deco` architecture, called `deco 2`, has been designed. This target architecture can be created using the `SCOTCH_archBuild2` routine; see Section 8.5.2, page 86 for further information. For further information on the rationale and implementation of these two features, please refer to [50].

Also,a new labeled tree-leaf architecture has been created, for nodes that label cores in non increasing order. See Section 6.4.2, page 28 for the description of the `ltleaf` target architecture.

Direct k-way graph partitioning and static mapping methods are now available. They are less expensive than the classical dual recursive bipartitioning scheme, and improve quality on average for numbers of parts above a few hundreds. Another new method aims at reducing load imbalance in the case of source graphs with highly irregular vertex weights; see Section 3.3, page 14. Users willing to keep using the old recursive bipartitioning strategies of the 5.x branch can create default strategies with the `SCOTCH_STRATRECURSIVE` flag set, in addition to other flags; see Section 8.3.1, page 65 for further information.

Graph repartitioning and static re-mapping features are now available; see Sections 8.8.2 to 8.8.8, starting from page 114.

The clustering capabilities of Scotch can be used more easily from the command line and library calls ; see Section 7.4.6 and Section 8.16.2.

A new set of routines has been created in order to compute vertex-separated, k-way partitions, that balance the loads of the parts and of the separator vertices that surround them; see Sections 8.3.4 and 8.8.6.

A method for computing independently orderings on connected components of a graph is now available; see Section 8.3.5, page 75, and the `SCOTCH_STRAT DISCONNECTED` flag in Section 8.3.1, page 65.

Key algorithms are now multi-threaded. See the installation file `INSTALL.txt` in the main directory for instructions on how to compile Scotch with thread support enabled.

Memory footprint measurement routines are now available to users; see Section 8.23, page 183.

The MeTiS compatibility library now provides optionally version 5 of the MeTiS API, in addition to version 3. In the `6.0` branch of Scotch, version 3 of the API will remain the default. To expose version 5 of the MeTiS API, Scotch must be compiled with flag `SCOTCH_METIS_VERSION=5` set. In further branches, version 5 of the API may become the default, requiring the `SCOTCH_METIS_VERSION=3` flag to be set to expose version 3 of the MeTiS API.

5.4 Changes in version 5.1 from version 5.0

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 9.3 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 patch level 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.

6 Files and data structures

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

6.1 Graph files

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

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.

Since version 3.3 has been introduced a new file format, referred to as the "new-style" file format, which replaces the previous, "old-style", file format. The two advantages of the new-style format over its predecessor are its greater compacity, which results in shorter I/O times, and its ability to handle easily graphs output by C or by Fortran programs.

Starting from version 4.0, only the new format is supported. To convert remaining old-style graph files into new-style graph files, one should get version 3.4 of the SCOTCH distribution, which comprises the `scv` file converter, and use it to produce new-style SCOTCH graph files from the old-style SCOTCH graph files which it is able to read. See section 7.4.5 for a description of `gcv`, formerly called `scv`.

The first line of a graph file holds the graph file version number, which is currently `0`. The second line holds the number of vertices of the graph (referred to as `vertnbr` in LIBSCOTCH; see for instance Figure 17, page 60, for a detailed example), followed by its number of arcs (unappropriately called `edgenbr`, as it is in fact equal to twice the actual number of edges). The third 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 [26], 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 graph base index.

This header data is then followed by as many lines as there are vertices in the graph, that is, `vertnbr` 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 4 shows the contents of a graph file modeling a cube with unity vertex and edge weights and base 0.

6.2 Mesh files

Mesh files, which usually end in "`.msh`", describe valuated meshes, made of elements and nodes, the elements of which can be mapped onto target architectures, and the nodes of which can be reordered.

Meshes are bipartite graphs, in the sense that every element is connected to the nodes that it comprises, and every node is connected to the elements to which it belongs. No edge connects any two element vertices, nor any two node vertices. One can also think of meshes as hypergraphs, such that nodes are the vertices of the hypergraph and elements are hyper-edges which connect multiple nodes, or

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

Figure 4: Graph file representing a cube.

reciprocally such that elements are the vertices of the hypergraph and nodes are hyper-edges which connect multiple elements.

Since meshes are graphs, the structure of mesh files resembles very much the one of graph files described above in section 6.1, and differs only by its header, which indicates which of the vertices are node vertices and element vertices.

The first line of a mesh file holds the mesh file version number, which is currently `1`. Graph and mesh version numbers will always differ, which enables application programs to accept both file formats and adapt their behavior according to the type of input data. The second line holds the number of elements of the mesh (`velmnbr`), followed by its number of nodes (`vnodnbr`), and its overall number of arcs (`edgenbr`, that is, twice the number of edges which connect elements to nodes and vice-versa).

The third line holds three figures: the base index of the first element vertex in memory (`velmbas`), the base index of the first node vertex in memory (`vnodbas`), and a numeric flag.

The SCOTCH mesh file format requires that all nodes and all elements be assigned to contiguous ranges of indices. Therefore, either all element vertices are defined before all node vertices, or all node vertices are defined before all element vertices. The node and element base indices indicate at the same time whether elements or nodes are put in the first place, as well as the value of the starting index used to describe the graph. Indeed, if `velmbas` $<$ `vnodbas`, then elements have the smallest indices, `velmbas` is the base value of the underlying graph (that is, `baseval` $=$ `velmbas`), and `velmbas` $+$ `velmnbr` $=$ `vnodbas` holds. Conversely, if `velmbas` $>$ `vnodbas`, then nodes have the smallest indices, `vnodbas` is the base value of the underlying graph, (that is, `baseval` $=$ `vnodbas`), and `vnodbas` $+$ `vnodnbr` $=$ `velmbas` holds.

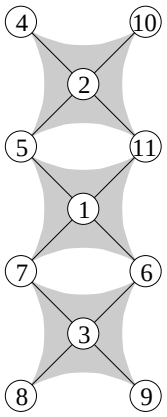
The numeric flag, similar to the one used by the CHACO graph format [26], is made of three decimal digits. A non-zero value in the units indicates that vertex weights are provided. A non-zero value in the tenths indicates that edge weights are provided. A non-zero value in the hundredths indicates that vertex labels are provided; if it is the case, and if `velmbas` $<$ `vnodbas` (resp. `velmbas` $>$ `vnodbas`), the `velmnbr` (resp. `vnodnbr`) first vertex lines are assumed to be element (resp. node) vertices, irrespective of their vertex labels, and the `vnodnbr` (resp. `velmnbr`) remaining vertex lines are assumed to be node (resp. element) vertices; else, natural order is assumed, starting at the underlying graph base index (`baseval`).

This header data is then followed by as many lines as there are node and element vertices in the graph. These lines are similar to the ones of the graph format, except

that, in order to save disk space, the numberings of nodes and elements all start from the same base value, that is, min(`velmbas`, `vnodbas`) (also called `baseval`, like for regular graphs).

For example, Figure 5 shows the contents of the mesh file modeling three square elements, with unity vertex and edge weights, elements defined before nodes, and numbering of the underlying graph starting from 1. In memory, the three elements are labeled from 1 to 3, and the eight nodes are labeled from 4 to 11. In the file, the three elements are still labeled from 1 to 3, while the eight nodes are labeled from 1 to 8.

When labels are used, elements and nodes may have similar labels, but not two elements, nor two nodes, should have the same labels.




```
1
3       8          24
1       4          000
4       2 (= 5)    8 (= 11)    4 (= 7)    3 (= 6)
4       7 (= 10)   2 (= 5)     8 (= 11)   1 (= 4)
4       5 (= 8)    6 (= 9)     3 (= 6)    4 (= 7)
1       2
2       2          1
2       1          3
2       1          3
1       3
1       3
1       2
2       2          1
```

Figure 5: Mesh file representing three square elements, with unity vertex and edge weights. Elements are defined before nodes, and numbering of the underlying graph starts from 1. The left part of the figure shows the mesh representation in memory, with consecutive element and node indices. The right part of the figure shows the contents of the file, with both element and node numberings starting from 1, the minimum of the element and node base values. Corresponding node indices in memory are shown in parentheses for the sake of comprehension.

6.3 Geometry files

Geometry files, which usually end in "`.xyz`", hold the coordinates of the vertices of their associated graph or mesh. These files are not used in the mapping process itself, since only topological properties are taken into account then (mappings are computed regardless of graph geometry). They are used by visualization programs to compute graphical representations of mapping results.

The first string to appear in a geometry file codes for its type, or dimensionality. It is "`1`" if the file contains unidimensional coordinates, "`2`" for bidimensional coordinates, and "`3`" for tridimensional coordinates. It is followed by the number of coordinate data stored in the file, which should be at least equal to the number of vertices of the associated graph or mesh, and by that many coordinate lines. Each coordinate line holds the label of the vertex, plus one, two or three real numbers which are the (X), (X,Y), or (X,Y,Z), coordinates of the graph vertices, according to the graph dimensionality.

Vertices can be stored in any order in the file. Moreover, a geometry file can have more coordinate data than there are vertices in the associated graph or mesh file; only coordinates the labels of which match labels of graph or mesh vertices will be taken into account. This feature allows all subgraphs of a given graph or mesh to share the same geometry file, provided that graph vertex labels remain unchanged. For example, Figure 6 shows the contents of the 3D geometry file associated with the graph of Figure 4.

```
3
8
0       0.0     0.0     0.0
1       0.0     0.0     1.0
2       0.0     1.0     0.0
3       0.0     1.0     1.0
4       1.0     0.0     0.0
5       1.0     0.0     1.0
6       1.0     1.0     0.0
7       1.0     1.0     1.0
```

Figure 6: Geometry file associated with the graph file of Figure 4.

6.4 Target files

Target files describe the architectures onto which source graphs are mapped. Instead of containing the structure of the target graph itself, as source graph files do, target files define how target graphs are bipartitioned and give the distances between all pairs of vertices (that is, processors). Keeping the bipartitioning information within target files avoids recomputing it every time a target architecture is used. We are allowed to do so because, in our approach, the recursive bipartitioning of the target graph is fully independent with respect to that of the source graph (however, the opposite is false).

For space and time saving issues, some classical homogeneous architectures (2D and 3D meshes and tori, hypercubes, complete graphs, etc.) have been algorithmically coded within the mapper itself by the means of built-in functions. Instead of containing the whole graph decomposition data, their target files hold only a few values, used as parameters by the built-in functions.

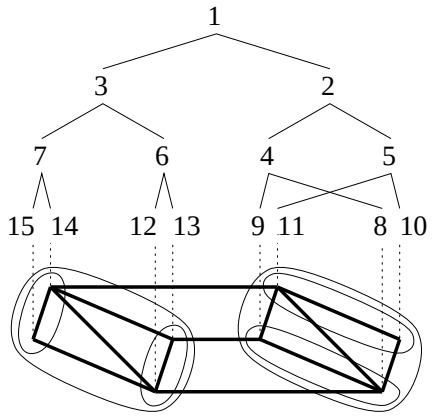
6.4.1 Decomposition-defined architecture files

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

Two main file formats coexist: the "`deco 0`" and "`deco 2`" formats. "`deco`" stands for "decomposition-defined architecture", followed by the format number. The "`deco 1`" format is a compiled form of the "`deco 0`" format, which we will not describe here as it is not meant to be handled by users.

The "`deco 0`" header is followed by two integer numbers, which are the number of processors and the largest terminal number used in the decomposition, respectively. Two arrays follow. The first array has as many lines as there are processors. Each of these lines holds three numbers: the processor label, the processor weight (that is an estimation of its computational power), and its terminal number. The terminal number associated with every processor is obtained by giving the initial domain holding all the processors number 1, and by numbering the two subdomains of a given domain of number i with numbers $2i$ and $2i + 1$. The second array is

a lower triangular diagonal-less matrix that gives the distance between all pairs of processors. This distance matrix, combined with the decomposition tree coded by terminal numbers, allows the evaluation by averaging of the distance between all pairs of domains. In order for the mapper to behave properly, distances between processors must be strictly positive numbers. Therefore, null distances are not accepted. For instance, Figure 7 shows the contents of the architecture decomposition file for UB(2, 3), the binary de Bruijn graph of dimension 3, as computed by the `amk_grf` program.




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

Figure 7: Target decomposition file for UB(2, 3). The terminal numbers associated with every processor define a unique recursive bipartitioning of the target graph.

The "`deco 2`" format was created so as to represent bigger target architectures. Indeed, the distance matrix of the "`deco 0`" format is quadratic in the number of target vertices, which is not scalable and prevents users from representing target architectures bigger than a few thousand vertices. In the "`deco 2`" architecture, distances are computed using in a multilevel representation of the target graph, in the form of a family of coarser graphs. Hence, the more distant the vertices are, the coarsest is the graph to be used to estimate this distance [50]. The vertices and edges of these graphs encode their respective cost of traversal, which becomes less accurate as coarser graphs are used.

6.4.2 Algorithmically-coded architecture files

Almost all algorithmically-coded architectures are defined with unity edge and vertex weights. They start with an abbreviation name of the architecture, followed by parameters specific to the architecture. The available built-in architecture definitions are listed below.

`cmplt` *size*
: Defines a complete graph with *size* vertices. Its vertex labels are numbers between 0 and $size - 1$.

`cmpltw` *size* $load_0$ $load_1$ ... $load_{size-1}$
: Defines a balanced weighted complete graph with *size* vertices. Its vertex labels are numbers between 0 and $size - 1$, and vertices are assigned integer

weights in the order in which these are provided. Subdomains are created in a way that aims to balance weights across target domain bipartitions.

`cmpltws` *size* $load_0$ $load_1$ ... $load_{size-1}$
: Defines a weighted complete graph with *size* vertices. Its vertex labels are numbers between 0 and $size - 1$, and vertices are assigned integer weights in the order in which these are provided. Target domains are kept in order during bipartitions.

`hcub` *dim*
: Defines a binary hypercube of dimension *dim*. Graph vertices are numbered according to the value of the binary representation of their coordinates in the hypercube.

`ltleaf` *levlnbr* $sizeval_0$ $linkval_0$ ... $sizeval_{levlnbr-1}$ $linkval_{levlnbr-1}$ *permnbr* $permval_0$ ... $permval_{permnbr-1}$
: The `ltleaf` (for "*labeled tree-leaf*") architecture is an extended tree-leaf architecture (`tleaf`, see below) which models target topologies where cores are not labeled in increasing order.
The tree structure of the architecture is described just like for a regular `tleaf` architecture. *permnbr* is the length of the permutation that is used to label cores, followed by this number of permutation indices, ranging between 0 and $(permnbr - 1)$. Figure 8 presents an example of such an architecture.
The permutation array must be of a size that matches level boundaries. Alternatively, a permutation of size 1, with only index 0 given, represents the identity permutation. In this case, the regular `tleaf` architecture can be used.

```
ltleaf
3 32 10 2 5 4 1
8 0 2 4 6 1 3 5 7
```

Figure 8: Labeled tree-leaf architecture with 3 levels, representing a system with 32 nodes of 2 quad-core processors. Inter-node communication costs 10, inter-processor communication within the same node costs 5 and inter-core communication within the same processor costs 1. Within a 8-core node, cores are labeled such that cores 0, 2, 4 and 6 are located on the first processor, while cores 1, 3, 5 and 7 are located on the second processor.

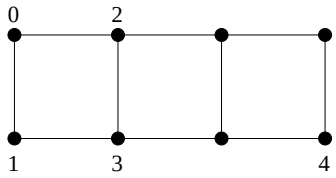
`mesh2D` dim_X dim_Y
: Defines a bidimensional array of dim_X columns by dim_Y rows. The vertex with coordinates (pos_X, pos_Y) has label $pos_X + pos_Y \times dim_X$.

`mesh3D` dim_X dim_Y dim_Z
: Defines a tridimensional array of dim_X columns by dim_Y rows by dim_Z levels. The vertex with coordinates (pos_X, pos_Y, pos_Z) has label $pos_X + pos_Y\, dim_X + pos_Z\, dim_X\, dim_Y$.

`meshXD` *ndims* dim_0 dim_1 ... $dim_{(ndims-1)}$
: Generalization of the `mesh2D` and `mesh3D` architectures. Defines a *ndims*-dimensional array of dimensions dim_0, dim_1 ... $dim_{ndims-1}$. The vertex with coordinates $(pos_0, pos_1, \ldots, pos_{ndims-1})$ has label $pos_0 + \sum_{d=1}^{ndims-1} \left(pos_d \prod_{d'=0}^{d-1} dim_{d'} \right)$.

`sub` *termnbr termnum*$_0$ *termnum*$_1$ ... *termnum*$_{termnbr-1}$ *architecture*

Defines a sub-architecture of another *architecture*. The sub-architecture contains *termnbr* vertices, which have ranks *termnum*$_0$, *termnum*$_1$, ... *termnum*$_{termnbr-1}$ in the prescribed, original *architecture*. The original architecture must comprise at least *termnbr* vertices, and thus cannot be a variable-sized architecture. The order in which vertex numbers are provided defines the part indices that will be used as output mapping data. For instance, in the example shown in Figure 9, source vertices that are assigned to vertex 3 of the sub-architecture are in fact assigned to vertex 5 of the original, 2D mesh architecture, according to its canonical numbering.

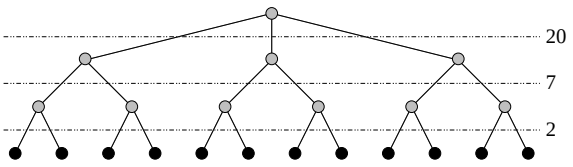



```
sub
5 0 4 1 5 7
mesh2D 4 2
```

Figure 9: Sub-architecture of a 4x2 `mesh2D` 2D grid architecture. The sub-architecture comprises 5 vertices, numbered from 0 to 4, which correspond to vertices 0, 4, 1, 5 and 7 of the original architecture, respectively.

`tleaf` *levlnbr sizeval*$_0$ *linkval*$_0$ ... *sizeval*$_{levlnbr-1}$ *linkval*$_{levlnbr-1}$

Defines a hierarchical, tree-shaped, architecture with *levlnbr* levels and $\sum_{i=0}^{levlnbr-1} sizeval_i$ leaf vertices. This topology is used to model hierarchical NUMA or NUIOA machines. The mapping is only computed with respect to the leaf vertices, which represent processing elements, while the upper levels of the tree model interconnection networks (intra-chip buses, inter-chip interconnection networks, network routers, etc.), as exemplified in Figure 10. The communication cost between two nodes is the cost of the highest common ancestor level.




```
tleaf
3 3 20 2 7 2 2
```

Figure 10: A "tree-leaf" graph with three levels. Processors are drawn in black and routers in grey. It has 3 levels, the first level has 3 sons and a traversal cost of 20, the second level has 2 sons and a traversal cost of 7, and the third level has also 2 sons and a traversal cost of 2.

`torus2D` *dim*$_X$ *dim*$_Y$

Defines a bidimensional array of *dim*$_X$ columns by *dim*$_Y$ rows, with wraparound edges. The vertex with coordinates (pos_X, pos_Y) has label $pos_X + pos_Y \times dim_X$.

`torus3D` dim_X dim_Y dim_Z

Defines a tridimensional array of dim_X columns by dim_Y rows by dim_Z levels, with wraparound edges. The vertex with coordinates (pos_X, pos_Y, pos_Z) has label $pos_X + pos_Y\, dim_X + pos_Z\, dim_X\, dim_Y$.

`torusXD` *ndims* dim_0 dim_1 ... $dim_{ndims-1}$

Generalization of the `torus2D` and `torus3D` architectures. Defines a *ndims*-dimensional torus of dimensions dim_0, dim_1 ... $dim_{ndims-1}$. The vertex with coordinates $(pos_0, pos_1, \ldots, pos_{(ndims-1)})$ has label $pos_0 + \sum_{d=1}^{ndims-1} \left(pos_d \prod_{d'=0}^{d-1} dim_{d'} \right)$.

6.4.3 Variable-sized architecture files

Variable-sized architectures are a class of algorithmically-coded architectures the size of which is not defined *a priori*. Domains of these target architectures can always be bipartitioned, again and again (until integer overflow occurs in domain indices). These architectures are used to perform graph clustering (see Sections 7.4.6 and 8.8.1), using a specifically tailored graph mapping strategy (see for instance Section 8.16.2).

As for fixed-size algorithmically-coded architectures, they start with an abbreviation name of the architecture, followed by parameters specific to the architecture. The available built-in variable-sized architecture definitions are listed below.

`varcmplt`

Defines a variable-sized complete graph. Domains are labeled such that the first domain is labeled 1, and the two subdomains of any domain i are labeled $2i$ and $2i + 1$. The distance between any two subdomains i and j is 0 if $i = j$ and 1 else.

`varhcub`

Defines a variable-sized hypercube. Domains are labeled such that the first domain is labeled 1, and the two subdomains of any domain i are labeled $2i$ and $2i + 1$. The distance between any two domains is the Hamming distance between the common bits of the two domains, plus half of the absolute difference between the levels of the two domains, this latter term modeling the average distance on unknown bits. For instance, the distance between subdomain $9 = 1001_B$, of level 3 (since its leftmost 1 has been shifted left thrice), and subdomain $53 = 110101_B$, of level 5 (since its leftmost 1 has been shifted left five times), is equal to 2: it is 1, which is the number of bits which differ between 1101_B (that is, $53 = 110101_B$ shifted rightwards twice) and 1001_B, plus 1, which is half of the absolute difference between 5 and 3.

6.5 Mapping files

Mapping files, which usually end in "`.map`", contain the result of the mapping of source graphs onto target architectures. They associate a vertex of the target graph with every vertex of the source graph.

Mapping files begin with the number of mapping lines which they contain, followed by that many mapping lines. Each mapping line holds a mapping pair, made of two integer numbers which are the label of a source graph vertex and the label of the target graph vertex onto which it is mapped. Mapping pairs can be stored in any order in the file; however, labels of source graph vertices must be all different. For example, Figure 11 shows the result obtained when mapping the source

graph of Figure 4 onto the target architecture of Figure 7. This one-to-one embedding of H(3) into UB$(2, 3)$ has dilation 1, except for one hypercube edge which has dilation 3.

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

Figure 11: Mapping file obtained when mapping the hypercube source graph of Figure 4 onto the binary de Bruijn architecture of Figure 7.

Mapping files are also used on output of the block orderer to represent the allocation of the vertices of the original graph to the column blocks associated with the ordering. In this case, column blocks are labeled in ascending order, such that the number of a block is always greater than the ones of its predecessors in the elimination process, that is, its leaves in the elimination tree.

6.6 Ordering files

Ordering files, which usually end in "`.ord`", contain the result of the ordering of source graphs or meshes that represent sparse matrices. They associate a number with every vertex of the source graph or mesh.

The structure of ordering files is analogous to the one of mapping files; they differ only by the meaning of their data.

Ordering files begin with the number of ordering lines which they contain, that is the number of vertices in the source graph or the number of nodes in the source mesh, followed by that many ordering lines. Each ordering line holds an ordering pair, made of two integer numbers which are the label of a source graph or mesh vertex and its rank in the ordering. Ranks range from the base value of the graph or mesh (`baseval`) to the base value plus the number of vertices (resp. nodes), minus one (`baseval` + `vertnbr` − 1 for graphs, and `baseval` + `vnodnbr` − 1 for meshes). Ordering pairs can be stored in any order in the file; however, indices of source vertices must be all different.

For example, Figure 12 shows the result obtained when reordering the source graph of Figure 4.

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

Figure 12: Ordering file obtained when reordering the hypercube graph of Figure 4.

The advantage of having both graph and mesh orderings start from `baseval` (and not `vnodbas` in the case of meshes) is that an ordering computed on the nodal

graph of some mesh has the same structure as an ordering computed from the native mesh structure, allowing for greater modularity. However, in memory, permutation indices for meshes are numbered from `vnodbas` to `vnodbas` + `vnodnbr` − 1.

6.7 Vertex list files

Vertex lists are used by programs that select vertices from graphs.

Vertex lists are coded as lists of integer numbers. The first integer is the number of vertices in the list and the other integers are the labels of the selected vertices, given in any order. For example, Figure 13 shows the list made from three vertices of labels 2, 45, and 7.

```
3   2   45   7
```

Figure 13: Example of vertex list with three vertices of labels 2, 45, and 7.

7 Programs

The programs of the SCOTCH project belong to five distinct classes.

- Graph handling programs, the names of which begin in "`g`", that serve to build and test source graphs.
- Mesh handling programs, the names of which begin in "`m`", that serve to build and test source meshes.
- Target architecture handling programs, the names of which begin in "`a`", that allow the user to build and test decomposition-defined target files, and especially to turn a source graph file into a target file.
- The mapping and ordering programs themselves.
- Output handling programs, which are the mapping performance analyzer, the graph factorization program, and the graph, matrix, and mapping visualization program.

The general architecture of the SCOTCH project is displayed in Figure 14.

7.1 Invocation

The programs comprising the SCOTCH project 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.

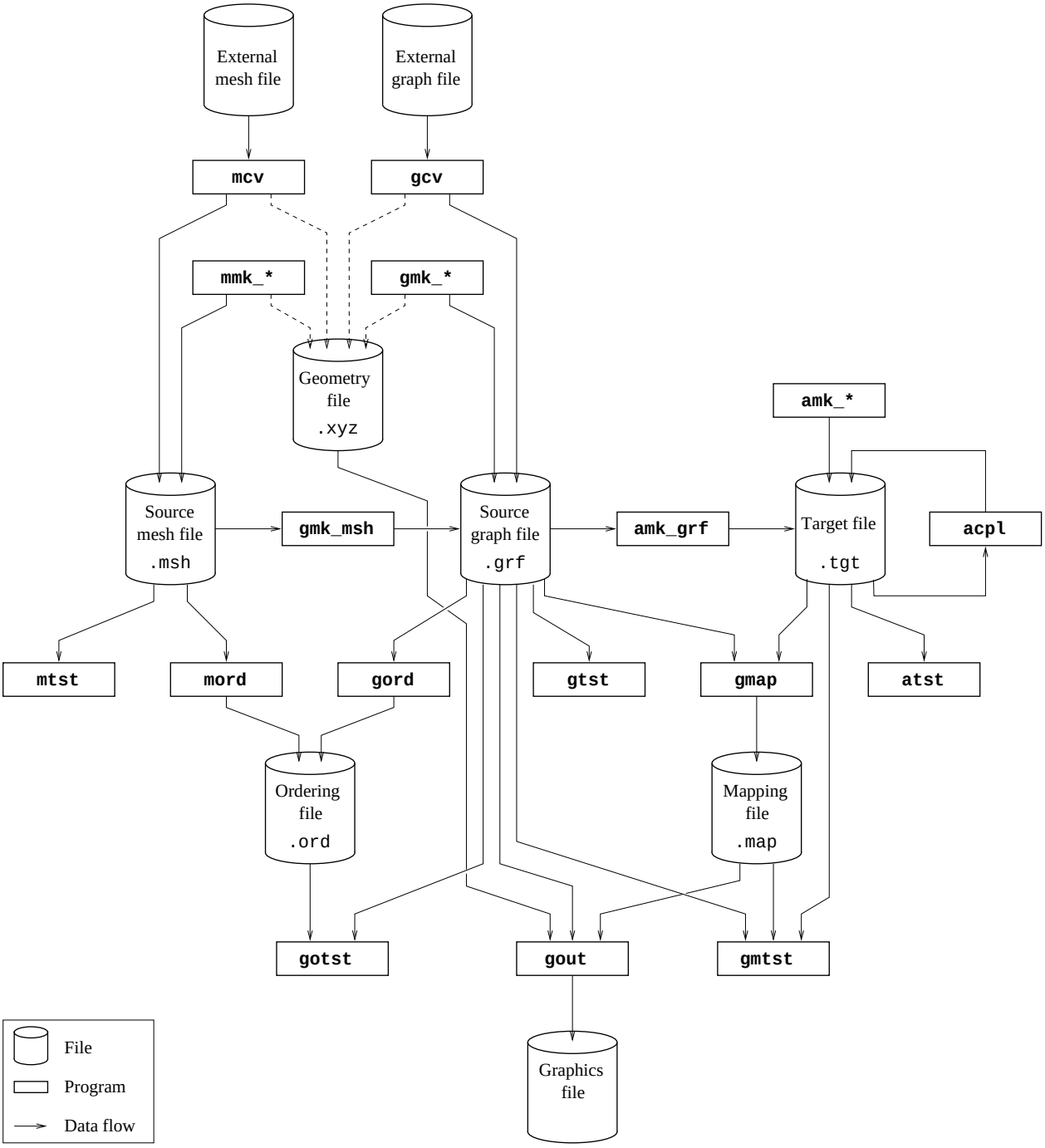


Figure 14: General architecture of the SCOTCH project. All of the features offered by the stand-alone programs are also available in the LIBSCOTCH library.

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 SCOTCH project are provided in section 10.

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 6, and use the `gtst` and `mtst` programs to check the consistency of source graphs and meshes.

7.2 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.

7.3 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 9.

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.

7.4 Description

7.4.1 acpl

Synopsis

`acpl` [*input_target_file* [*output_target_file*]] *options*

Description

The program `acpl` is the decomposition-defined architecture file compiler. It processes architecture files of type "`deco 0`" built by hand or by the `amk_*` programs, to create a "`deco 1`" compiled architecture file of about four times the size of the original one; see section 6.4.1, page 27, for a detailed description of decomposition-defined target architecture file formats.
The mapper can read both original and compiled architecture file formats. However, compiled architecture files are read much more efficiently, as they are directly loaded into memory without further processing. Since the compilation time of a target architecture graph evolves as the square of its number of vertices, precompiling with `acpl` can save some time when many mappings are to be performed onto the same large target architecture.

Options

`-h` Display the program synopsis.

`-V` Print the program version and copyright.

7.4.2 amk_*

Synopsis

`amk_ccc` *dim* [*output_target_file*] *options*

`amk_fft2` *dim* [*output_target_file*] *options*

`amk_hy` *dim* [*output_target_file*] *options*

`amk_m2` *dimX* [*dimY* [*output_target_file*]] *options*

`amk_p2` *weight0* [*weight1* [*output_target_file*]] *options*

Description

The `amk_*` programs make target graphs. Each of them is devoted to a

specific topology, for which it builds target graphs of any dimension.
These programs are an alternate way between algorithmically-coded built-in target architectures and decompositions computed by mapping with `amk_grf`. Like built-in target architectures, their decompositions are algorithmically computed, and like `amk_grf`, their output is a decomposition-defined target architecture file. These programs allow the definition and testing of new algorithmically-coded target architectures without coding them in the core of the mapper.

Program `amk_ccc` outputs the target architecture file of a Cube-Connected-Cycles graph of dimension *dim*. Vertex (l, m) of CCC(dim), with $0 \leq l < dim$ and $0 \leq m < 2^{dim}$, is linked to vertices $((l-1) \bmod dim, m)$, $((l+1) \bmod dim, m)$, and $(l, m \oplus 2^l)$, and is labeled $l \times 2^{dim} + m$. $\oplus$ denotes the bitwise exclusive-or binary operator, and $a \bmod b$ the integer remainder of the euclidian division of a by b.

Program `amk_fft2` outputs the target architecture file of a binary Fast-Fourier-Transform graph of dimension *dim*. Vertex (l, m) of FFT(dim), with $0 \leq l \leq dim$ and $0 \leq m < 2^{dim}$, is linked to vertices $(l-1, m)$, $(l-1, m \bmod 2^{l-1})$, $(l+1, m)$, and $(l+1, m \oplus 2^l)$, if they exist, and is labeled $l \times 2^{dim} + m$.

Program `amk_hy` outputs the target architecture file of a hypercube graph of dimension *dim*. Vertices are labeled according to the decimal value of their binary representation. The decomposition-defined target architectures computed by `amk_hy` do not exactly give the same results as the built-in hypercube targets because distances are not computed in the same manner, although the two recursive bipartitionings are identical. To achieve best performance and save space, use the built-in architecture.

Program `amk_p2` outputs the target architecture file of a weighted path graph with two vertices, the weights of which are given as parameters.
This simple target topology is used to bipartition a source graph into two weighted parts with as few cut edges as possible. In particular, it is used to compute independent partitions of the processors of a multi-user parallel machine. As a matter of fact, if the yet unallocated part of the machine is represented by a source graph with n vertices, and n' processors are requested by a user in order to run a job (with $n' \leq n$), mapping the source graph onto the weighted path graph with two vertices of weights n' and $n - n'$ leads to a partition of the machine in which the allocated n' processors should be as densely connected as possible (see Figure 15).

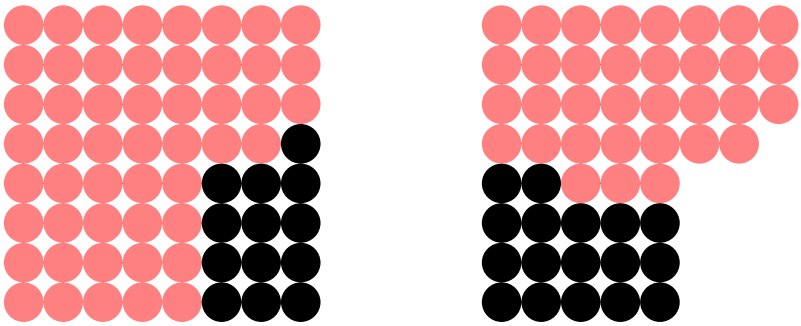
Options

`-h` Display the program synopsis.

`-m`*method*
Select the bipartitioning method (for `amk_m2` only).

`n` Nested dissection.

`o` Dimension-per-dimension one-way dissection. This is less efficient than nested dissection, and this feature exists only for benchmarking purposes.



a. Construction of a partition with 13 vertices (in black) on a 8×8 bidimensional mesh architecture.

b. Construction of a partition with 17 vertices (in black) on the remaining architecture.

Figure 15: Construction of partitions on a bidimensional 8×8 mesh architecture by weighted bipartitioning.

`-V` Print the program version and copyright.

7.4.3 `amk_grf`

Synopsis

`amk_grf` [*input_graph_file* [*output_target_file*]] *options*

Description

The program `amk_grf` turns a source graph file into a decomposition-defined target architecture file.

The `-2` option creates a "`deco 2`" decomposition rather than a "`deco 0`" one. See Section 6.4.1, page 27 for more information on the different types of decomposition-defined target architectures.

The `-l` option restricts the target architecture to the vertices indicated in the given vertex list file. It is therefore possible to build a target architecture made of several disconnected parts of a bigger architecture. Note that this is not equivalent to turning a disconnected source graph into a target architecture, since doing so would lead to an architecture made of several independent pieces at infinite distance one from another. Considering the selected vertices within their original architecture makes it possible to compute the distance between vertices belonging to distinct connected components, and therefore to evaluate the cost of the mapping of two neighbor processes onto disjoint areas of the architecture.
The restriction feature is very useful in the context of multi-user parallel machines. On these machines, when users request processors in order to run their jobs, the partitions allocated by the operating system may not be regular nor connected, because of existing partitions already attributed to other people. By feeding `amk_grf` with the source graph representing the whole parallel machine, and the vertex list containing the labels of the processors allocated by the operating system, it is possible to build a target architecture corresponding to this partition, and therefore to map processes on it, automatically, regardless of the partition shape.

The -b option selects the recursive bipartitioning strategy used to build the "deco 0" decomposition of the source graph. For regular, unweighted, topologies, the '-b(g|h)fx' recursive bipartitioning strategy should work best. For irregular or weighted graphs, use the default strategy, which is more flexible. See also the manual page of function SCOTCH_archBuild0, page 85, for further information.

Options

-b*strategy*
: Use recursive bipartitioning strategy *strategy* to build the decomposition of the architecture graph. The format of bipartitioning strategies is defined within section 8.3.3, at page 69.

-h
: Display the program synopsis.

-l*input_vertex_file*
: Load vertex list from *input_vertex_file*. As for all other file names, "-" may be used to indicate standard input.

-V
: Print the program version and copyright.

7.4.4 atst

Synopsis

atst [*input_target_file* [*output_data_file*]] *options*

Description

The program atst is the architecture tester. It gives some statistics on decomposition-defined target architectures, and in particular the minimum, maximum, and average communication costs (that is, weighted distance) between all pairs of processors.

Options

-h
: Display the program synopsis.

-V
: Print the program version and copyright.

7.4.5 gcv

Synopsis

gcv [*input_graph_file* [*output_graph_file* [*output_geometry_file*]]] *options*

Description

The program gcv is the source graph converter. It takes on input a graph file of the format specified with the -i option, and outputs its equivalent in the format specified with the -o option, along with its associated geometry file whenever geometry data is available. At the time being, it accepts four input formats: the Matrix Market format [6], the Harwell-Boeing collection format [11], the Chaco/MeTiS graph format [26], and the Scotch format. Three output format are available: the Matrix Market format, the Chaco/MeTiS graph format and the Scotch source graph and geometry data format.

Options

`-h` Display the program synopsis.

`-i`*format*
Specify the type of input graph. The available input formats are listed below.

`b`[*number*]
Harwell-Boeing graph collection format. Only symmetric assembled matrices are currently supported. Since files in this format can contain several graphs one after another, the optional integer *number*, starting from 0, indicates which graph of the file is considered for conversion.

`c` Chaco v1.0/MeTiS format.

`m` The Matrix Market format.

`s` Scotch source graph format.

`-o`*format*
Specify the output graph format. The available output formats are listed below.

`c` Chaco v1.0/MeTiS format.

`m` The Matrix Market format.

`s` Scotch source graph format.

`-V` Print the program version and copyright.

Default option set is "`-Ib0 -Os`".

7.4.6 `gmap` / `gpart`

Synopsis

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

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

Description

The program `gmap` is the graph mapper. It uses a partitioning strategy to map a source graph onto a target graph, so that the weight of source graph vertices allocated to target vertices is balanced, and the communication cost function f_C is minimized.

The program `gpart` is the graph partitioner. It uses a partitioning strategy to split a source graph into the prescribed number of parts, using vertex or edge separators, depending whether the `-o` option is set or not.

The implemented mapping methods mainly derive from graph theory. In particular, graph geometry is never used, even if it is available; only topological properties are taken into account. Mapping methods are used to define mapping strategies by means of selection, combination, grouping, and condition operators.

Mapping methods implemented in version 6.0 comprise direct k-way methods, including a k-way multilevel framework and k-way local refinement methods, as well as the Dual Recursive Bipartitioning algorithm, which uses graph bipartitioning methods. Available bipartitioning methods include a multilevel framework that uses other bipartitioning methods to compute the initial and refined bipartitions: an improved implementation of the Fiduccia–Mattheyses heuristic designed to handle weighted graphs, a diffusion-based algorithm, a greedy method derived from the Gibbs, Poole, and Stockmeyer algorithm, a greedy graph growing heuristic, a greedy "exactifying" refinement algorithm designed to balance vertex loads as much as possible, etc.

`gpart` is a simplified interface to `gmap`, 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.

Both programs can be used to perform clustering, by means of the `-q` option. `gpart` will perform topology-independent clustering, while `gmap` may compute locality-preserving clusters when mapping onto variable-sized, non-complete, architectures (see Section 6.4.3).

If mapping statistics are wanted rather than the mapping output itself, mapping output can be set to `/dev/null`, with option `-vmt` to get mapping statistics and timings.

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 execution context options according to the given flags. Some of these flags are antagonistic, while others can be combined. Default values depend on compilation flags `COMMON_RANDOM_FIXED_SEED` and `SCOTCH_DETERMINISTIC`.

  `d` Enforce a deterministic behavior across multiple runs, even in a multi-threaded context. Determinism may slow down execution for some multi-threaded algorithms, as determinism requires more synchronization across threads. Implies '`f`'.

  `f` Use a fixed random seed. Suffices to enforce determinism in a single-threaded context.

  `r` Use a variable random seed for each run. Opposite of '`f`'.

  `u` Allow for undeterministic behavior across multiple runs. May result in faster execution in a multi-threaded context. Opposite of '`d`'.

`-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 8.3.1 for more information. The currently available flags are the following.

- `b` Enforce load balance as much as possible.
- `q` Privilege quality over speed.
- `r` Only use recursive bipartitioning methods.
- `s` Privilege speed over quality.
- `t` Use only safe methods in the strategy.

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 mapping strategy *strat*. In the case of static mapping or of edge-based graph partitioning, the format of mapping strategies should comply with the format defined in Section 8.3.2. If the `-o` option is used (see below), strategies must be vertex partitioning strategies, which are described in Section 8.3.4. This option is incompatible with options `-b` and `-c`.

`-o` Compute vertex-based partitions rather than static mappings or edge-based partitions. This option is only valid for `gpart`, or when `gmap` is called with a target architecture which is an unweighted complete graph.

`-q` (for `gpart`)

`-q`*pwght*
(for `gmap`) Perform clustering instead of partitioning or mapping. Clustering is achieved by means of a specific strategy string that performs recursive bipartitioning until the size of the parts is smaller than some threshold value. For `gpart`, this value replaces the desired number of parts as the first argument passed to the program. For `gmap`, the threshold must be given just after the `-q` option.

`-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. For a detailed description of the data displayed, please refer to the manual page of `gmtst` below.

- `m` Mapping or partitioning information, depending whether the `-o` option has been set or not.
- `s` Strategy information. This parameter displays the mapping strategy which will be used by `gmap` or `gpart`.
- `t` Timing information.

`-V` Print the program version and copyright.

7.4.7 gmk_*

Synopsis

gmk_hy *dim* [*output_graph_file*] *options*

gmk_m2 *dimX* [*dimY* [*output_graph_file*]] *options*

gmk_m3 *dimX* [*dimY* [*dimZ* [*output_graph_file*]]] *options*

gmk_ub2 *dim* [*output_graph_file*] *options*

Description

The gmk_* programs make source graphs. Each of them is devoted to a specific topology, for which it builds target graphs of any dimension.
The gmk_* programs are mainly used in conjunction with amk_grf. Most gmk_* programs build source graphs describing parallel machines, which are used by amk_grf to generate corresponding target sub-architectures, by means of its -l option. Such a procedure is shown in section 10, which builds a target architecture from five vertices of a binary de Bruijn graph of dimension 3.

Program gmk_hy outputs the source file of a hypercube graph of dimension *dim*. Vertices are labeled according to the decimal value of their binary representation.

Program gmk_m2 outputs the source file of a bidimensional grid with *dimX* columns and *dimY* rows. If the -t option is set, tori are built instead of grids. The vertex of coordinates $(posX, posY)$ is labeled $posY \times dimX + posX$.

Program gmk_m3 outputs the source file of a tridimensional grid with *dimZ* layers of *dimY* rows by *dimX* columns. If the -t option is set, tori are built instead of grids. The vertex of coordinates $(posX, posY)$ is labeled $(posZ \times dimY + posY) \times dimX + posX$.

Program gmk_ub2 outputs the source file of a binary unoriented de Bruijn graph of dimension *dim*. Vertices are labeled according to the decimal value of their binary representation.

Options

-b*base_value*
: Set the base value of the produced graph.

-e Build a 8-neighbor grid rather than a 4-neighbor grid (for gmk_m2 only).

-g*output_geometry_file*
: Output graph geometry to file *output_geometry_file* (for gmk_m* only). As for all other file names, "-" may be used to indicate standard output.

-h Display the program synopsis.

-t Build a torus rather than a grid (for gmk_m* only).

-V Print the program version and copyright.

`-y` Invert the y coordinate in the produced geometry (for `gmk_m*` only). This may be useful to display 2D grids in the PostScript format, using the `gout` program, since in this case the y axis is oriented downwards.

7.4.8 gmk_msh

Synopsis

`gmk_msh` [*input_mesh_file* [*output_graph_file*]] *options*

Description

The `gmk_msh` program builds a graph file from a mesh file. All of the nodes of the mesh are turned into graph vertices, and edges are created between all pairs of vertices that share an element (that is, elements are turned into cliques).

Options

`-h` Display the program synopsis.

`-V` Print the program version and copyright.

7.4.9 gmtst

Synopsis

`gmtst` [*input_graph_file* [*input_target_file* [*input_mapping_file* [*output_data_file*]]]] *options*

Description

The program `gmtst` is the graph mapping tester. It outputs some statistics on the given mapping, regarding load balance and inter-processor communication.
The two first statistics lines deal with process mapping statistics, while the following ones deal with communication statistics. The first mapping line gives the number of processors used by the mapping, followed by the number of processors available in the architecture, and the ratio of these two numbers, written between parentheses. The second mapping line gives the minimum, maximum, and average loads of the processors, followed by the variance of the load distribution, and an imbalance ratio equal to the maximum load over the average load. The first communication line gives the minimum and maximum number of neighbors over all blocks of the mapping, followed by the sum of the number of neighbors over all blocks of the mapping, that is the total number of messages that have to be sent to exchange data between all neighboring blocks. The second communication line gives the average dilation of the edges, followed by the sum of all edge dilations. The third communication line gives the average expansion of the edges, followed by the value of function f_C. The fourth communication line gives the average cut of the edges, followed by the number of cut edges. The fifth communication line shows the ratio of the average expansion over the average dilation; it is smaller than 1 when the mapper succeeds in putting heavily intercommunicating processes closer to each other than it does for lightly communicating processes; it is equal to 1 if all edges

have the same weight. The remaining lines form a distance histogram, which shows the amount of communication load that involves processors located at increasing distances.

`gmtst` allows the testing of cross-architecture mappings. By inputing it a target architecture different from the one that has been used to compute the mapping, but with compatible vertex labels, one can see what the mapping would yield on this new target architecture.

Options

`-h` Display the program synopsis.

`-V` Print the program version and copyright.

7.4.10 `gord`

Synopsis

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

Description

The `gord` program is the block sparse matrix graph 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 nested dissection method [19], but classical [42] and state-of-the-art [1, 49] minimum degree algorithms are implemented as well. Ordering methods are used to define ordering strategies by means of selection, grouping, and condition operators.

For the nested dissection method, vertex separation methods comprise algorithms that directly compute vertex separators, as well as methods that build vertex separators from edge separators, *i.e.* graph bipartitions (all of the graph bipartitioning methods available in the static mapper `gmap` can be used in this latter case).

The `-o` option allows the user to define the ordering strategy. The `-c` option allows the user to set preferences on the behavior of the ordering strategy which is used by default.

When the graphs to order are very large, the same results can be obtained by using the `dgord` parallel program of the PT-Scotch distribution, which can read centralized graph files too.

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 execution context options according to the given flags. Some of these flags are antagonistic, while others can be combined. Default

values depend on compilation flags `COMMON_RANDOM_FIXED_SEED` and `SCOTCH_DETERMINISTIC`.

- `d` Enforce a deterministic behavior across multiple runs, even in a multi-threaded context. Determinism may slow down execution for some multi-threaded algorithms, as determinism requires more synchronization across threads. Implies '`f`'.
- `f` Use a fixed random seed. Suffices to enforce determinism in a single-threaded context.
- `r` Use a variable random seed for each run. Opposite of '`f`'.
- `u` Allow for undeterministic behavior across multiple runs. May result in faster execution in a multi-threaded context. Opposite of '`d`'.

`-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 8.3.1 for more information. The resulting strategy string can be displayed by means of the `-vs` option.

- `b` Enforce load balance as much as possible.
- `d` Handle separately connected components, in case the graph is disconnected.
- `q` Privilege quality over speed. This is the default behavior.
- `s` Privilege speed over quality.
- `t` Use only safe methods in the strategy.

`-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 given in section 6.5.

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 ordering strategy *strat*. The format of ordering strategies is defined in section 8.3.5.

`-t` *output_tree_file*

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

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

`-V` Print the program version and copyright.

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

`s` Strategy information. This parameter displays the ordering strategy which will be used by `gord`.

`t` Timing information.

7.4.11 gotst

Synopsis

`gotst` [*input_graph_file* [*input_ordering_file* [*output_data_file*]]] *options*

Description

The program `gotst` is the ordering tester. It gives some statistics on orderings, including the number of non-zeros and the operation count of the factored matrix, as well as statistics regarding the elimination tree. Since it performs the factorization of the reordered matrix, it can take a very long time and consume a large amount of memory when applied to large graphs. The first two statistics lines deal with the elimination tree. The first one displays the number of leaves, while the second shows the minimum height of the tree (that is, the length of the shortest path from any leaf to the –or a– root node), its maximum height, its average height, and the variance of the heights with respect to the average. The third line displays the number of non-zero terms in the factored matrix, the amount of index data that is necessary to maintain the block structure of the factored matrix, and the number of operations required to factor the matrix by means of Cholesky factorization.

Options

`-h` Display the program synopsis.

`-v` Do not account for vertex weights when computing factorization costs.

`-V` Print the program version and copyright.

7.4.12 gout

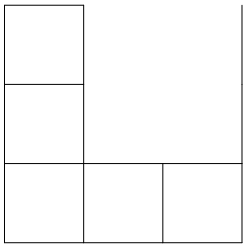
Synopsis

`gout` [*input_graph_file* [*input_geometry_file* [*input_mapping_file* [*output_visualization_file*]]]] *options*

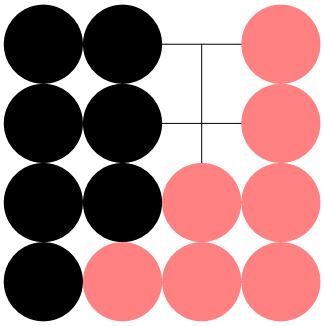
Description

The `gout` program is the graph, matrix, and mapping viewer program. It takes on input a source graph, its geometry file, and optionally a mapping result file, and produces a file suitable for display. At the time being, `gout` can generate plain and encapsulated PostScript files for the display of adjacency matrix patterns and the display of planar graphs (although tridimensional objects can be displayed by means of isometric projection, the display of tridimensional mappings is not efficient), and Open Inventor [57], Tulip [4], and VTK files for the interactive visualization of tridimensional graphs.

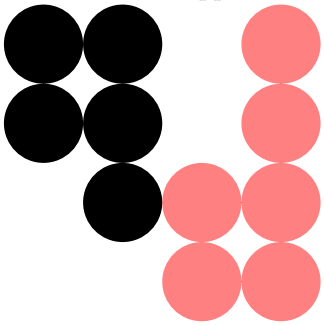
In the case of mapping display, the number of mapping pairs contained in the input mapping file may differ from the number of vertices of the input source graph; only mapping pairs the source labels of which match labels of source graph vertices will be taken into account for display. This feature allows the user to show the result of the mapping of a subgraph drawn on the whole graph, or else to outline the most important aspects of a mapping by restricting the display to a limited portion of the graph. For example, Figure 16.b shows how the result of the mapping of a subgraph of the bidimensional mesh $M_2(4,4)$ onto the complete graph $K(2)$ can be displayed on the whole $M_2(4,4)$ graph, and Figure 16.c shows how the display of the same mapping can be restricted to a subgraph of the original graph.



a. A subgraph of $M_2(4,4)$ to be mapped onto $K(2)$.



b. Mapping result displayed on the full $M_2(4,4)$ graph.



c. Mapping result displayed on another subgraph of $M_2(4,4)$.

Figure 16: PostScript diplay of a single mapping file with different subgraphs of the same source graph. Vertices covered with disks of the same color are mapped onto the same processor.

Options

-g parameters

Geometry parameters.

n Do not read geometry data. This option can be used in conjunction with option `-om` to avoid reading the geometry file when displaying the pattern of the adjacency matrix associated with the source graph, since geometry data are not needed in this case. If this option is set, the geometry file is not read. However, if an *output_visualization_file* name is given in the command line, dummy *input_geometry_file* and

input_mapping_file names must be specified so that the file argument count is correct. In this case, use the "–" parameter to take standard input as a dummy geometry input stream. In practice, the `-om` and `-gn` options always imply the `-mn` option.

`r` For bidimensional geometry only, rotate geometry data by 90 degrees, counter-clockwise.

`-h` Display the program synopsis.

`-mn`

Do not read mapping data, and display the graph without any mapping information. If this option is set, the mapping file is not read. However, if an *output_visualization_file* name is given in the command line, a dummy *input_mapping_file* name must be specified so that the file argument count is correct. In this case, use the "–" parameter to take standard input as a dummy mapping input stream.

`-o`*format*[{*parameters*}]

Specify the type of output, with optional parameters within curly braces and separated by commas. The output formats are listed below.

`i` Output the graph in SGI's Open Inventor format, in ASCII mode, suitable for display by the `ivview` program [57]. The optional parameters are given below.

- `c` Color output, using 16 different colors. Opposite of `g`.
- `g` Grey-level output, using 8 different levels. Opposite of `c`.
- `r` Remove cut edges. Edges the ends of which are mapped onto different processors are not displayed. Opposite of `v`.
- `v` View cut edges. All graph edges are displayed. Opposite of `r`.

`m` Output the pattern of the adjacency matrix associated with the source graph, in Adobe's PostScript format. The optional parameters are given below.

- `e` Encapsulated PostScript output, suitable for LaTeX use with `epsf`. Opposite of `f`.
- `f` Full-page PostScript output, suitable for direct printing. Opposite of `e`.

`p` Output the graph in Adobe's PostScript format. The optional parameters are given below.

- `a` Avoid displaying the mapping disks. Opposite of `d`.
- `c` Color PostScript output, using 16 different colors. Opposite of `g`.
- `d` Display the mapping disks. Opposite of `a`.
- `e` Encapsulated PostScript output, suitable for LaTeX use with `epsf`. Opposite of `f`.
- `f` Full-page PostScript output, suitable for direct printing. Opposite of `e`.
- `g` Grey-level PostScript output. Opposite of `c`.
- `l` Large clipping. Mapping disks are included in the clipping area computation. Opposite of `s`.
- `r` Remove cut edges. Edges the ends of which are mapped onto different processors are not displayed. Opposite of `v`.

s Small clipping. Mapping disks are excluded from the clipping area computation. Opposite of `l`.

v View cut edges. All graph edges are displayed. Opposite of `r`.

`x=`*val*
Minimum X relative clipping position (in [0.0;1.0]).

`X=`*val*
Maximum X relative clipping position (in [0.0;1.0]).

`y=`*val*
Minimum Y relative clipping position (in [0.0;1.0]).

`Y=`*val*
Maximum Y relative clipping position (in [0.0;1.0]).

`t` Output the graph in the TULIP graph format, suitable for display by the TULIP data visualization software [4]. The optional parameters are given below.

a Avoid displaying mapping spheres around vertices. Opposite of `d`.

b Black-and-white output. Opposite of `c`.

c Color output, using 16 different colors. Opposite of `b`.

d Display mapping spheres around vertices. Opposite of `a`.

r Remove cut edges. Edges the ends of which are mapped onto different processors are not displayed. Opposite of `v`.

v View cut edges. All graph edges are displayed. Opposite of `r`.

`v` Output the graph in VTK legacy ASCII format, suitable for display by the `paraview` program [38]. The graph partition is represented as an integer scalar dataset called `mapValues`. Unmapped vertices are assigned to part index `0`, while higher part indices represent regular parts (hence, part number i of some mapping becomes part index $i+1$ in the VTK dataset). The optional parameters are given below.

r Remove cut edges. Edges the ends of which are mapped onto different processors are not displayed. Opposite of `v`.

v View cut edges. All graph edges are displayed. Opposite of `r`.

`-V` Print the program version and copyright.

Default option set is "`-Oi{v}`".

7.4.13 gtst

Synopsis

`gtst` [*input_graph_file* [*output_data_file*]] *options*

Description

The program `gtst` 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.

When the graphs to test are very large, the same results can be obtained by using the `dgtst` parallel program of the PT-Scotch distribution, which can read centralized graph files too.

Options

`-h` Display the program synopsis.

`-V` Print the program version and copyright.

7.4.14 mcv

Synopsis

`mcv` [*input_mesh_file* [*output_mesh_file* [*output_geometry_file*]]] *options*

Description

The program `mcv` is the source mesh converter. It takes on input a mesh file of the format specified with the `-i` option, and outputs its equivalent in the format specified with the `-o` option, along with its associated geometry file whenever geometrical data is available. At the time being, it only accepts one external input format: the Harwell-Boeing format [11], for square elemental matrices only. The only output format to date is the Scotch source mesh and geometry data format.

Options

`-h` Display the program synopsis.

`-i`*format*
: Specify the type of input mesh. The available input formats are listed below.

  `b`[*number*]
  : Harwell-Boeing mesh collection format. Only symmetric elemental matrices are currently supported. Since files in this format can contain several meshes one after another, the optional integer *number*, starting from 0, indicates which mesh of the file is considered for conversion.

  `s` Scotch source mesh format.

`-o`*format*
: Specify the output graph format. The available output formats are listed below.

  `s` Scotch source graph format.

`-V` Print the program version and copyright.

Default option set is "`-Ib0 -Os`".

7.4.15 mmk_*

Synopsis

`mmk_m2` *dimX* [*dimY* [*output_mesh_file*]] *options*

`mmk_m3` *dimX* [*dimY* [*dimZ* [*output_mesh_file*]]] *options*

Description

The `mmk_*` programs make source meshes.

Program `mmk_m2` outputs the source file of a bidimensional mesh with $dimX \times dimY$ elements and $(dimX + 1) \times (dimY + 1)$ nodes. The element of coordinates $(posX, posY)$ is labeled $posY \times dimX + posX$.

Program `mmk_m3` outputs the source file of a tridimensional mesh with $dimX \times dimY \times dimZ$ elements and $(dimX + 1) \times (dimY + 1) \times (dimZ + 1)$ nodes.

Options

`-g`*output_geometry_file*
: Output mesh geometry to file *output_geometry_file* (for `mmk_m2` only). As for all other file names, "-" may be used to indicate standard output.

`-h` Display the program synopsis.

`-V` Print the program version and copyright.

7.4.16 mord

Synopsis

`mord` [*input_mesh_file* [*output_ordering_file* [*output_log_file*]]] *options*

Description

The `mord` program is the block sparse matrix mesh orderer. It uses an ordering strategy to compute block orderings of sparse matrices represented as source meshes, whose node vertex weights indicate the number of DOFs per node (if this number is non homogeneous), 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 nested dissection method [19], but classical [42] and state-of-the-art [1, 49] minimum degree algorithms are implemented as well. Ordering methods are used to define ordering strategies by means of selection, grouping, and condition operators.

The `-o` option allows the user to define the ordering strategy. The `-c` option allows the user to set preferences on the behavior of the ordering strategy which is used by default.

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 8.3.1 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.

-h Display the program synopsis.

-m*output_mapping_file*

Write to *output_mapping_file* the mapping of mesh node 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 leaves in the elimination tree. The structure of mapping files is given in section 6.5.

When the coordinates of the node vertices are available, the mapping file may be processed by program `gout`, along with the graph structure that can be created from the source mesh file by means of the `gmk_msh` program, to display the node vertex separators and supervariable amalgamations that have been computed.

-o*strat*

Apply ordering strategy *strat*. The format of ordering strategies is defined in section 8.3.5.

-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 node vertex index. This integer number is the number of the column block which is the parent of the column block to which the node vertex belongs, or -1 if the column block to which the node vertex belongs is a root of the separator tree (there can be several roots, if the mesh is disconnected).

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

-V Print the program version and copyright.

-v*verb*

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

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

t Timing information.

7.4.17 mtst

Synopsis

`mtst` [*input_mesh_file* [*output_data_file*]] *options*

Description

The program `mtst` is the source mesh tester. It checks the consistency of the input source mesh structure (matching of arcs that link elements to nodes and nodes to elements, number of elements, nodes, and edges, etc.), and gives some statistics regarding element and node weights, edge weights, and element and node degrees.

Options

`-h` Display the program synopsis.

`-V` Print the program version and copyright.

8 Library

All of the features provided by the programs of the SCOTCH distribution may be directly accessed by calling the appropriate functions of the libSCOTCH library, archived in files `libscotch.a` and `libscotcherr.a`. These routines belong to six distinct classes:

- source graph and source mesh handling routines, which serve to declare, build, load, save, and check the consistency of source graphs and meshes, along with their geometry data;
- target architecture handling routines, which allow the user to declare, build, load, and save target architectures;
- strategy handling routines, which allow the user to declare and build mapping and ordering strategies;
- mapping routines, which serve to declare, compute, and save mappings of source graphs to target architectures by means of mapping strategies;
- a partitioning-with-overlap routine, which computes a vertex separator that splits a graph into a prescribed number of parts, such that the vertex load of each part and of its neighboring separator vertices are balanced;
- ordering routines, which allow the user to declare, compute, and save orderings of source graphs and meshes;
- error handling routines, which allow the user either to provide his own error servicing routines, or to use the default routines provided in the libSCOTCH distribution.

A METIS compatibility library, called `libscotchmetis.a`, is also available. It allows users who were previously using METIS in their software to take advantage of the efficieny of SCOTCH without having to modify their code. The services provided by this library are described in Section 8.24.

8.1 Calling the routines of libSCOTCH

8.1.1 Calling from C

All of the C routines of the libSCOTCH library are prefixed with "`SCOTCH_`". The remainder of the function names is made of the name of the type of object to which

the functions apply (e.g. "graph", "mesh", "arch", "map", 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 a stream, 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_graphInit and SCOTCH_graphLoad routines, described in sections 8.7.16 and 8.7.17, respectively, can be called from C by using the following code.

```
#include <stdio.h>
#include "scotch.h"
 ...
 SCOTCH_Graph grafdat;
 FILE * fileptr;

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

Since "scotch.h" uses several system objects which are declared in "stdio.h", this latter file 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 "scotch.h" file as the include file for compiling a program which uses only the sequential routines of the libScotch library.

8.1.2 Calling from Fortran

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

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

For routines which 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 which 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_graphInit` and `SCOTCH_graphLoad` routines, described in sections 8.7.16 and 8.7.17, respectively, can be called from Fortran by using the following code.

```
      INCLUDE "scotchf.h"
      DOUBLEPRECISION GRAFDAT(SCOTCH_GRAPHDIM)
      INTEGER RETVAL
      ...
      CALL SCOTCHFGRAPHINIT (GRAFDAT (1), RETVAL)
      IF (RETVAL .NE. 0) THEN
      ...
      OPEN (10, FILE='brol.grf')
      CALL SCOTCHFGRAPHLOAD (GRAFDAT (1), FNUM (10), 1, 0, RETVAL)
      CLOSE (10)
      IF (RETVAL .NE. 0) THEN
      ...
```

Although the "`scotchf.h`" and "`ptscotchf.h`" files may look very similar on your system, never mistake them, and always use the "`scotchf.h`" file as the include file for compiling a program which uses only the sequential routines of the libScotch library.

8.1.3 Compiling and linking

The compilation of C or Fortran routines which use routines of the libScotch library requires that either "`scotch.h`" or "`scotchf.h`" be included, respectively.

The routines of the libScotch library are grouped in a library file called `libscotch.a`. Default error routines that print an error message and exit are provided in library file `libscotcherr.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: "`-lscotch -lscotcherr -lm`". If you want to handle errors by yourself, you should not link with library file `libscotcherr.a`, but rather provide a `SCOTCH_errorPrint()` routine. Please refer to section 8.19 for more information.

Programs that call both sequential and parallel routines of Scotch should use only the parallel versions of the include file and of the library. Please refer to the equivalent section of the PT-Scotch user's manual for more information.

8.1.4 Dynamic library issues

The advantage of dynamic libraries is that application code may not need to be recompiled when the library is updated. Whether this is true or not depends on the extent of the changes. One of the cases when recompilation is mandatory is when API data structures change: code that statically reserves space for them may be subject to boundary overflow errors when the size of library data structures increase, so that library routines operate on more space than what was statically allocated by the compiler based on the header files of the old version of the library.

In order to alleviate this problem, the libScotch proposes a set of routines to dynamically allocate storage space for the opaque API Scotch structures. Because these routines return pointers, these `SCOTCH_*Alloc` routines, as well as the `SCOTCH_free` routine, are only available in the C interface. Alternately, the

SCOTCH_*Sizeof routines may be used to obtain dynamically the size of these opaque structures.

8.1.5 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 the one 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 9.3 for more information on the setting of these compilation flags).

Consequently, the C interface of SCOTCH uses two types of integers. Graph-related quantities are passed as SCOTCH_Nums, while system-related values such as file handles, as well as return values of libSCOTCH routines, are always passed as ints.

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 INTEGERs. 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 "-DINTSIZEx" compilation flags. Also, the INTEGER*idx type represents an integer type of a size equivalent to the one of a SCOTCH_Idx, as set by the "-DIDXSIZEx" 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_Nums, 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_Nums 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 ints 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 v3 library, as they are only available as a 32-bit implementation.

8.1.6 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. These algorithms are enabled when Scotch is compiled with the flag "-DSCOTCH_PTHREAD" set.

Unless explicitly prevented to do so, libScotch library routines will detect the number of cores available on the user's system and will use as many of them as prescribed at compile time or, if no upper threshold was set at that time, all of those which are currently available. This behavior can be controlled further by means of the shell environment variable "SCOTCH_PTHREAD_NUMBER=x", where x is the prescribed maximum number of threads to be used. Setting a thread number to 1 will coerce Scotch into using only purely sequential algorithms (which may differ in nature from their multi-threaded counterparts). Setting the thread number to -1 will make Scotch use all available cores, overriding the value possibly set at compile time.

Another way for users to control concurrency is to use SCOTCH_Context objects (see Section 8.21). These objects define user-configurable execution contexts, in which libScotch library routines can be executed independently from others. They allow the user to prescribe a given number of threads, as well as their binding to the available cores, or even to capture a pool of existing threads, to make them participate in the computation of libScotch library routines. They also allow the user to create private pseudo-random generators, so that concurrently executing library routines do not interfere with each other and produce fully reproducible results.

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.

8.2 Data types

All of the data used in the libScotch interface are of integer type SCOTCH_Num. To hide the internals of Scotch to callers, all of the data structures are opaque, that is, declared within "scotch.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). Scotch internally manages with base values and array pointers so as to process these arrays accordingly.

8.2.1 SCOTCH_Arch architecture type

Target architecture structures are completely opaque. The only way to describe an architecture is by means of a graph passed to the SCOTCH_archBuild or SCOTCH_archBuild2 routines.

8.2.2 SCOTCH_Graph graph type

Source graphs are described by means of adjacency lists. These data are stored in arrays and scalars of type SCOTCH_Num, as shown in Figures 17 and 18. The graph fields have the following meaning:

`baseval`
: Base value for all array indexing.

`vertnbr`
: Number of vertices in graph.

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

`verttab`
: Array of start indices in `edgetab` of vertex adjacency sub-arrays.

`vendtab`
: Array of after-last indices in `edgetab` of vertex adjacency sub-arrays. For any vertex i, with $\texttt{baseval} \leq i < (\texttt{baseval} + \texttt{vertnbr})$, $(\texttt{vendtab}[i] - \texttt{verttab}[i])$ is the degree of vertex i, and the indices of the neighbors of i are stored in `edgetab` from $\texttt{edgetab}[\texttt{verttab}[i]]$ to $\texttt{edgetab}[\texttt{vendtab}[i] - 1]$, inclusive.

  When all vertex adjacency lists are stored in order in `edgetab`, it is possible to save memory by not allocating the physical memory for `vendtab`. In this case, illustrated in Figure 17, `verttab` is of size $\texttt{vertnbr} + 1$ and `vendtab` points to $\texttt{verttab} + 1$. This case is referred to as the "compact edge array" case, such that `verttab` is sorted in ascending order, $\texttt{verttab}[\texttt{baseval}] = \texttt{baseval}$ and $\texttt{verttab}[\texttt{baseval} + \texttt{vertnbr}] = (\texttt{baseval} + \texttt{edgenbr})$.

`velotab`
: Optional array, of size `vertnbr`, holding the integer load associated with every vertex.

`edgetab`
: Array, of a size equal at least to $(\max_i(\texttt{vendtab}[i]) - \texttt{baseval})$, holding the adjacency array of every vertex.

`edlotab`
: Optional array, of a size equal at least to $(\max_i(\texttt{vendtab}[i]) - \texttt{baseval})$, holding the integer load associated with every arc. Matching arcs should always have identical loads.

Dynamic graphs can be handled elegantly by using the `vendtab` array. In order to dynamically manage graphs, one just has to allocate `verttab`, `vendtab` and `edgetab` arrays that are large enough to contain all of the expected new vertex and edge data. Original vertices are labeled starting from `baseval`, leaving free space at the end of the arrays. To remove some vertex i, one just has to replace $\texttt{verttab}[i]$ and $\texttt{vendtab}[i]$ with the values of $\texttt{verttab}[\texttt{vertnbr} - 1]$ and $\texttt{vendtab}[\texttt{vertnbr} - 1]$, respectively, and browse the adjacencies of all neighbors of former vertex $\texttt{vertnbr} - 1$ such that all $(\texttt{vertnbr} - 1)$ indices are turned into is. Then, `vertnbr` must be decremented, and `SCOTCH_graphBuild()` must be called to account for the change of topology. If a graph building routine such as `SCOTCH_graphLoad()` or `SCOTCH_graphBuild()` had already been called on the `SCOTCH_Graph` structure, `SCOTCH_graphFree()` 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 $\texttt{verttab}[\texttt{vertnbr} - 1]$ and $\texttt{vendtab}[\texttt{vertnbr} - 1]$ with the starting and end indices of the adjacency sub-array of the new vertex.

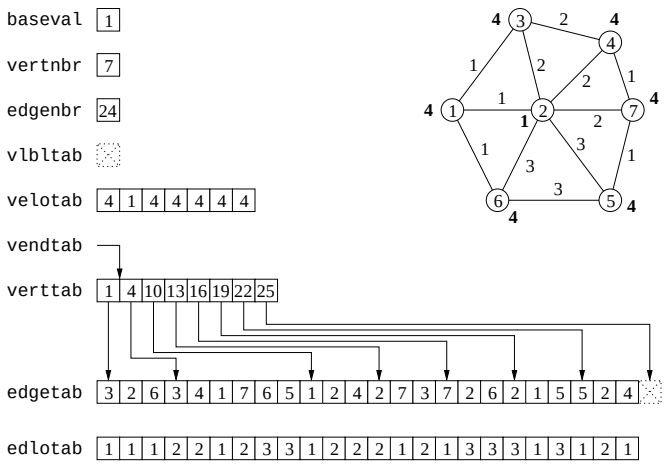


Figure 17: Sample graph and its description by LIBSCOTCH arrays using a compact edge array. Numbers within vertices are vertex indices, bold numbers close to vertices are vertex loads, and numbers close to edges are edge loads. Since the edge array is compact, `verttab` is of size `vertnbr`+1 and `vendtab` points to `verttab`+1.

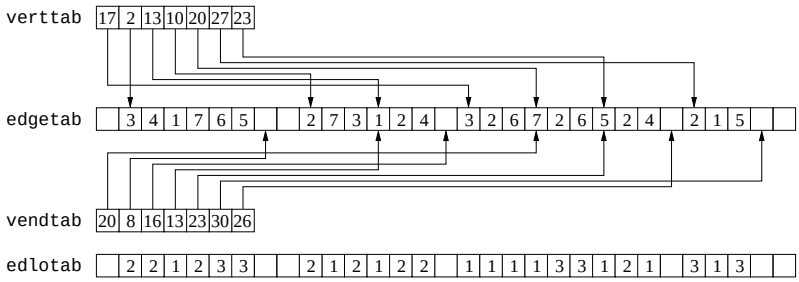


Figure 18: Adjacency structure of the sample graph of Figure 17 with disjoint edge and edge load arrays. Both `verttab` and `vendtab` are of size `vertnbr`. This allows for the handling of dynamic graphs, the structure of which can evolve with time.

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 `vendtab`[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 `verttab`[i] and `vendtab`[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.

8.2.3 `SCOTCH_Mesh` mesh type

Since meshes are basically bipartite graphs, source meshes are also described by means of adjacency lists. The description of a mesh requires several `SCOTCH_Num` scalars and arrays, as shown in Figure 19. They have the following meaning:

`velmbas`
: Base value for element indexings.

`vnodbas`
: Base value for node indexings. The base value of the underlying graph, `baseval`, is set as min(`velmbas`, `vnodbas`).

`velmnbr`
: Number of element vertices in mesh.

`vnodnbr`
: Number of node vertices in mesh. The overall number of vertices in the underlying graph, `vertnbr`, is set as `velmnbr` + `vnodnbr`.

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

`verttab`
: Array of start indices in `edgetab` of vertex (that is, both elements and nodes) adjacency sub-arrays.

`vendtab`
: Array of after-last indices in `edgetab` of vertex adjacency sub-arrays. For any element or node vertex i, with `baseval` $\leq i <$ (`baseval` + `vertnbr`), `vendtab`[i] − `verttab`[i] is the degree of vertex i, and the indices of the neighbors of i are stored in `edgetab` from `edgetab`[`verttab`[i]] to `edgetab`[`vendtab`[i] − 1], inclusive.

  When all vertex adjacency lists are stored in order in `edgetab`, it is possible to save memory by not allocating the physical memory for `vendtab`. In this case, illustrated in Figure 19, `verttab` is of size (`vertnbr` + 1) and `vendtab` points to (`verttab` + 1). This case is referred to as the "compact edge array" case, such that `verttab` is sorted in ascending order, `verttab`[`baseval`] = `baseval` and `verttab`[`baseval` + `vertnbr`] = (`baseval` + `edgenbr`).

`velotab`
: Array, of size `vertnbr`, holding the integer load associated with each vertex.

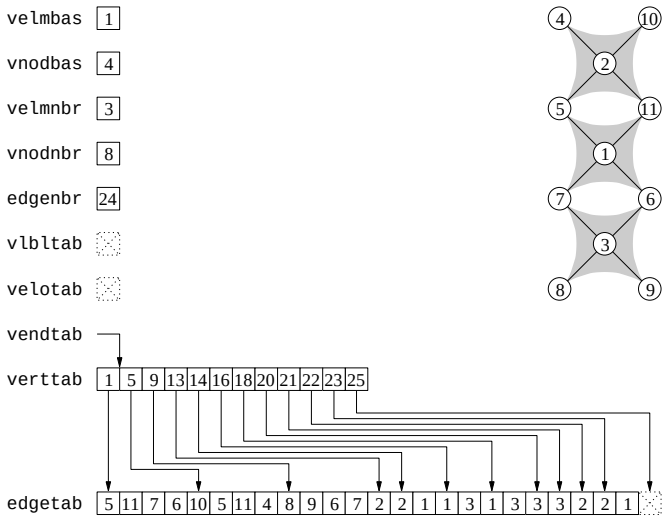


Figure 19: Sample mesh and its description by LIBSCOTCH arrays using a compact edge array. Numbers within vertices are vertex indices. Since the edge array is compact, `verttab` is of size (`vertnbr` + 1) and `vendtab` points to (`verttab` + 1).

As for graphs, it is possible to handle elegantly dynamic meshes by means of the `verttab` and `vendtab` arrays. There is, however, an additional constraint, which is that mesh nodes and elements must be ordered consecutively. The solution to fulfill this constraint in the context of mesh ordering is to keep a set of empty elements (that is, elements which have no node adjacency attached to them) between the element and node arrays. For instance, Figure 20 represents a 4-element mesh with 6 nodes, and such that 4 element vertex slots have been reserved for new elements and nodes. These slots are empty elements for which `verttab`[i] equals `vendtab`[i], irrespective of these values, since they will not lead to any memory access in `edgetab`.

Using this layout of vertices, new nodes and elements can be created by growing the element and node sub-arrays into the empty element sub-array, by both of its sides, without having to re-write the whole mesh structure, as illustrated in Figure 21. Empty elements are transparent to the mesh ordering routines, which base their work on node vertices only. Users who want to update the arrays of a mesh that has already been declared using the `SCOTCH_meshBuild()` routine must call `SCOTCH_meshExit()` prior to updating the mesh arrays, and then call `SCOTCH_meshBuild()` again after the arrays have been updated, so that the `SCOTCH_Mesh` structure remains consistent with the new mesh data.

8.2.4 `SCOTCH_Geom` geometry type

Geometry data is always associated with a graph or a mesh. It is simply made of a single array of double-precision values which represent the coordinates of the vertices of a graph, or of the node vertices of a mesh, in vertex order. The fields of a geometry structure are the following:

`dimnnbr`
: Number of dimensions of the graph or of the mesh, which can be 1, 2, or 3.

`geomtab`
: Array of coordinates. This is an array of double precision values organized as

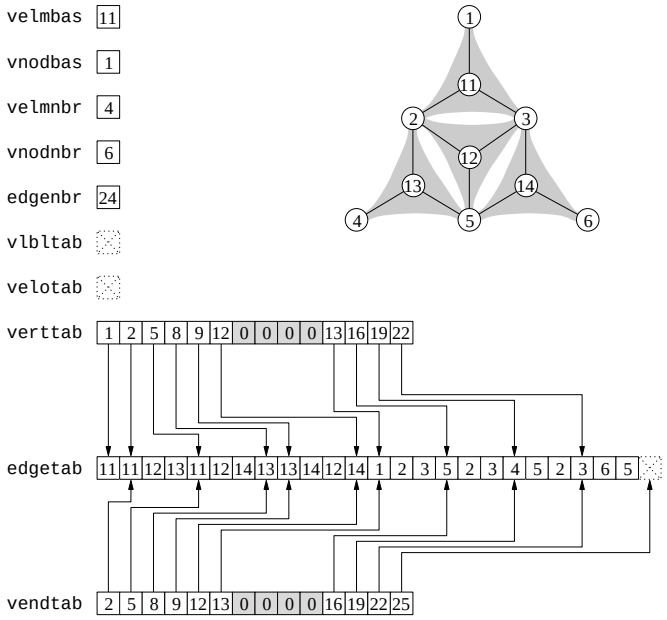


Figure 20: Sample mesh and its description by LIBSCOTCH arrays, with nodes numbered first and elements numbered last. In order to allow for dynamic re-meshing, empty elements (in grey) have been inserted between existing node and element vertices.

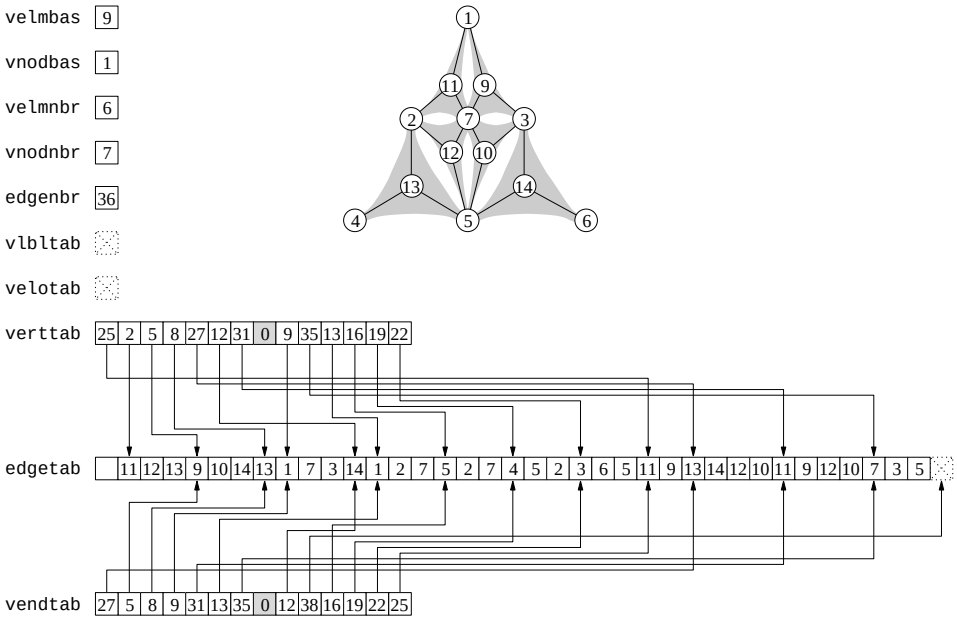


Figure 21: Re-meshing of the mesh of Figure 20. New node vertices have been added at the end of the vertex sub-array, new elements have been added at the beginning of the element sub-array, and vertex base values have been updated accordingly. Node adjacency lists that could not fit in place have been added at the end of the edge array, and some of the freed space has been re-used for new adjacency lists. Element adjacency lists do not require moving in this case, as all of the elements have the name number of nodes.

an array of (x), or (x, y), or (x, y, z) tuples, according to `dimnnbr`. Coordinates that are not used (e.g. the z coordinates for a bidimentional object) are not allocated. Therefore, the x coordinate of some graph vertex i is located at `geomtab[`$(i - $`baseval)` $* $`dimnnbr` $+$ `baseval]`, its y coordinate is located at `geomtab[`$(i - $`baseval)` $* $`dimnnbr` $+$ `baseval` $+ 1$`]` if `dimnnbr` ≥ 2, and its z coordinate is located at `geomtab[`$(i - $`baseval)` $* $`dimnnbr` $+$ `baseval` $+ 2$`]` if `dimnnbr` $= 3$. Whenever the geometry is associated with a mesh, only node vertices are considered, so the x coordinate of some mesh node vertex i, with `vnodbas` $\leq i$, is located at `geomtab[`$(i - $`vnodbas`)` $* $`dimnnbr` $+$ `baseval]`, its $y$ coordinate is located at `geomtab[`$(i - $`vnodbas`)` $* $`dimnnbr` $+$ `baseval` $+ 1$`]` if `dimnnbr` ≥ 2, and its z coordinate is located at `geomtab[`$(i - $`vnodbas`)` $* $`dimnnbr` $+$ `baseval` $+ 2$`]` if `dimnnbr` $= 3$.

8.2.5 `SCOTCH_Ordering` block ordering format

Block orderings associated with graphs and meshes are described by means of block and permutation arrays, made of `SCOTCH_Num`s, as shown in Figure 22. In order for all orderings to have the same structure, irrespective of whether they are created from graphs or meshes, all ordering data indices start from `baseval`, even when they refer to a mesh the node vertices of which are labeled from a `vnodbas` index such that `vnodbas` $>$ `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_Graph` structure if the ordering was computed from a graph, and with node vertex $i+$(`vnodbas` $-$ `baseval`) of the `SCOTCH_Mesh` structure if the ordering was computed from a mesh. 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. Indices in `rangtab` are based. Therefore, `rangtab[baseval]` is always equal to `baseval`, and `rangtab[cblknbr+baseval]` is always equal to `vertnbr`$+$`baseval` for graphs and to `vnodnbr` $+$ `baseval` for meshes. 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, `vertnbr` $+ 1$ for graphs and `vnodnbr` $+ 1$ for meshes.

`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 sub-blocks, as the block

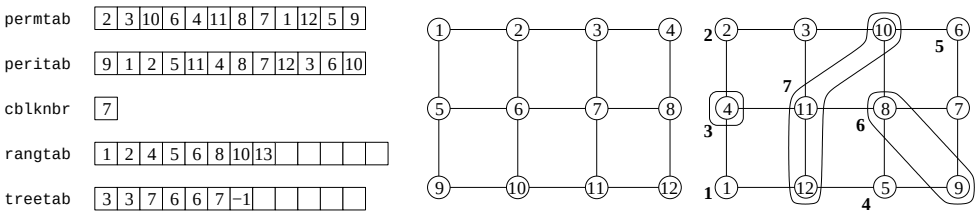


Figure 22: Arrays resulting from the ordering by complete nested dissection of a 4 by 3 grid based from 1. Leftmost grid is the original grid, and righmost grid is the reordered grid, with separators shown and column block indices written in bold.

splitting, minimum fill or minimum degree methods do, all sub-blocks 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 22 for a complete example.

8.3 Strategy strings

The behavior of the 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, or to meshes and submeshes.

8.3.1 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 released.

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 when 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` (see Section 8.16.2 and after), 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.

Mapping and partitioning flags

`SCOTCH_STRATRECURSIVE`
: Use only recursive bipartitioning methods, and not direct k-way methods. When this flag is not set, any combination of methods can be used, so as to achieve the best result according to other user preferences.

`SCOTCH_STRATREMAP`
: Use the strategy for remapping an existing partition.

Ordering flags

`SCOTCH_STRATDISCONNECTED`
: Find and handle independently disconnected components.

`SCOTCH_STRATLEVELMAX`
: Create at most the prescribed levels of nested dissection separators.

`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.

8.3.2 Mapping strategy strings

At the time being, mapping methods only apply to graphs, as there is not yet a mesh mapping tool in the Scotch package.

Mapping strategies are made of methods, with optional parameters enclosed between curly braces, and separated by commas, in the form of *method*[{*parameters*}] . The currently available mapping methods are the following.

`b` Band method. This method builds a band graph of given width around the current frontier of the k-way partition to which it is applied, and calls a graph mapping strategy to refine the equivalent k-way partition of the band graph. Then, the refined frontier of the band graph is projected back to the current graph. This method was initially presented in [9] in the case of bipartitioning. The parameters of the band bipartitioning method are listed below.

`bnd=`*strat*
: Set the graph mapping strategy to be used on the band graph.

`org=`*strat*
: Set the fallback graph mapping strategy to be used on the original graph if the band graph strategy could not be used. The three cases which require the use of this fallback strategy are the following. First, if the separator of the original graph is empty, which makes it impossible to compute a band graph. Second, if any part of the band graph to be built is of the same size as the one of the original graph. Third, if the application of the `bnd` bipartitioning method to the band graph leads to a situation where any two anchor vertices are placed in the same part.

`width=`*val*
: Set the width of the band graph. All graph vertices that are at a distance less than or equal to *val* from any frontier vertex are kept in the band graph.

`d` Diffusion method. This method, presented in [44] in the case of bipartitioning, flows k kinds of antagonistic liquids from k source vertices, and sets the new frontier as the limit between vertices which contain different kinds of liquids. Because selecting the source vertices is essential to the obtainment of useful results, this method has been hard-coded so that the k source vertices are the k 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 physical analogy of this method loses weight when it is applied to target architectures that are not complete graphs. Also, since this method does not guarantee load balance, it should not be used as the last term of a selection operator, as it is most likely that its outcome will never be selected; it should always be followed by a method that restores load balance, such as the Fiduccia-Mattheyses method. The parameters of the diffusion mapping 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.

`f` *k*-way Fiduccia-Mattheyses method. The parameters of the Fiduccia-Mattheyses method are listed below.

`bal`=*rat*
: Set the maximum weight imbalance ratio to the given fraction of the subgraph vertex weight. Common values are around 0.01, that is, one percent.

`move`=*nbr*
: Maximum number of hill-climbing moves that can be performed before a pass ends. During each of its passes, the Fiduccia-Mattheyses algorithm repeatedly swaps vertices between parts so as to minimize the cost function. A pass completes either when all of the vertices have been moved once, or if too many swaps that do not decrease the value of the cost function have been performed. Setting this value to zero turns the Fiduccia-Mattheyses algorithm into a gradient-like method, which may be used to quickly refine partitions during the uncoarsening phase of the multilevel method.

`pass`=*nbr*
: Set the maximum number of optimization passes performed by the algorithm. The Fiduccia-Mattheyses algorithm stops as soon as a pass has not yielded any improvement of the cost function, or when the maximum number of passes has been reached. Value -1 stands for an infinite number of passes, that is, as many as needed by the algorithm to converge.

`m` Multilevel method. The parameters of the multilevel method are listed below.

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

`low`=*strat*
: Set the strategy that is used to compute the mapping of the coarsest 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 vertices than the minimum number of vertices allowed.

`vert`=*nbr*
: Set the threshold under which graphs are no longer coarsened. Coarsening stops when either the coarsening ratio is above the maximum coarsening ratio, or the graph would have fewer vertices than the minimum number of vertices allowed. When the target architecture is a variable-sized architecture, coarsening stops when the coarsened graph would have less than *nbr* vertices. When the target architecture is a regular, fixed-size, architecture, coarsening stops when each subdomain would have less

than *nbr* vertices, that is, when the size of the coarsened graph would have less than *nbr* × `domnnbr` vertices, where `domnnbr` is the number of vertices in the target architecture.

`r` Dual Recursive Bipartitioning mapping algorithm, as defined in section 3.2. The parameters of the DRB mapping method are listed below.

`job=`*tie*
: The *tie* flag defines how new jobs are stored in job pools.

  `t` Tie job pools together. Subjobs are stored in same pool as their parent job. This is the default behavior, as it proves the most efficient in practice.

  `u` Untie job pools. Subjobs are stored in the next job pool to be processed.

`map=`*tie*
: The *tie* flag defines how results of bipartitioning jobs are propagated to jobs still in pools.

  `t` Tie both mapping tables together. Results are immediately available to jobs in the same job pool. This is the default behavior.

  `u` Untie mapping tables. Results are only available to jobs of next pool to be processed.

`poli=`*policy*
: Select jobs according to policy *policy*. Job selection policies define how bipartitioning jobs are ordered within the currently active job pool. Valid policy flags are

  `L` Most neighbors of higher level.

  `l` Highest level.

  `r` Random.

  `S` Most neighbors of smaller size. This is the default behavior.

  `s` Biggest size.

`sep=`*strat*
: Apply bipartitioning strategy *strat* to each bipartitioning job. A bipartitioning strategy is made of one or several bipartitioning methods, which can be combined by means of strategy operators. Graph bipartitioning strategies are described below.

`x` Exactifier method, as defined in Section 3.3. This greedy algorithm refines the current mapping so as to reduce load imbalance as much as possible. Since this method does not consider communication minimization, its use should be restricted to cases where achieving load balance is critical and where recursive bipartitioning may fail to achieve it, because of very irregular vertex loads.

8.3.3 Graph bipartitioning strategy strings

A graph bipartitioning strategy is made of one or several 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 separate 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 variable, *val* is either a graph variable or a constant of the type of variable *var*, and *relop* is one of '<', '=', and '>'. The graph variables are listed below, along with their types.

    `deg`
    : The average degree of the current graph. Float.

    `edge`
    : The number of arcs (which is twice the number of edges) of the current graph. Integer.

    `load`
    : The overall vertex load (weight) of the current graph. Integer.

    `load0`
    : The vertex load of the first subset of the current bipartition of the current graph. Integer.

    `vert`
    : The number of vertices of the current graph. Integer.

method[{*parameters*}]
: Bipartitioning method. For bipartitioning methods that can be parametrized, parameter settings may be provided after the method name. Parameters must be separated by commas, and the whole list be enclosed between curly braces.

The currently available graph bipartitioning methods are the following.

`b` Band method. This method builds a band graph of given width around the current frontier of the graph to which it is applied, and calls a graph bipartitioning strategy to refine the equivalent bipartition of the band graph. Then, the refined frontier of the band graph is projected back to the current graph. This method, presented in [9], was created to reduce the cost of vertex separator refinement algorithms in a multilevel context, but it improves partition quality too. The same behavior is observed for graph bipartitioning. The parameters of the band bipartitioning method are listed below.

`bnd=`*strat*
: Set the graph bipartitioning strategy to be used on the band graph.

`org=`*strat*
: Set the fallback graph bipartitioning strategy to be used on the original graph if the band graph strategy could not be used. The three cases which require the use of this fallback strategy are the following. First, if the separator of the original graph is empty, which makes it impossible to compute a band graph. Second, if any part of the band graph to be built is of the same size as the one of the original graph. Third, if the application of the `bnd` bipartitioning method to the band graph leads to a situation where both anchor vertices are placed in the same part.

`width=`*val*
: Set the width of the band graph. All graph vertices that are at a distance less than or equal to *val* from any frontier vertex are kept in the band graph.

`d` Diffusion method. This method, presented in [44], 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. Also, since this method does not guarantee load balance, it should not be used as the last term of a selection operator, as it is most likely that its outcome will never be selected; it should always be followed by a method that restores load balance, such as the Fiduccia-Mattheyses method. 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.

`f` Fiduccia-Mattheyses method. The parameters of the Fiduccia-Mattheyses method are listed below.

`bal`=*rat*
Set the maximum weight imbalance ratio to the given fraction of the subgraph vertex weight. Common values are around 0.01, that is, one percent.

`move`=*nbr*
Maximum number of hill-climbing moves that can be performed before a pass ends. During each of its passes, the Fiduccia-Mattheyses algorithm repeatedly swaps vertices between the two parts so as to minimize the cost function. A pass completes either when all of the vertices have been moved once, or if too many swaps that do not decrease the value of the cost function have been performed. Setting this value to zero turns the Fiduccia-Mattheyses algorithm into a gradient-like method, which may be used to quickly refine partitions during the uncoarsening phase of the multilevel method.

`pass`=*nbr*
Set the maximum number of optimization passes performed by the algorithm. The Fiduccia-Mattheyses algorithm stops as soon as a pass has not yielded any improvement of the cost function, or when the maximum number of passes has been reached. Value -1 stands for an infinite number of passes, that is, as many as needed by the algorithm to converge.

`a` Genetic algorithm method. This method has two parameters.

`pass`=*nbr*
Set the number of generations bred by the algorithm.

`pop`=*nbr*
Set the population count.

`g` Gibbs-Poole-Stockmeyer method. This method has only one parameter.

`pass`=*nbr*
Set the number of sweeps performed by the algorithm.

`h` Greedy-graph-growing method. This method has only one parameter.

`pass`=*nbr*
Set the number of runs performed by the algorithm.

`m` Multilevel method. The parameters of the multilevel method are listed below.

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

`low=`*strat*
: Set the strategy that is used to compute the partition of the coarsest 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 vertices than the minimum number of vertices allowed.

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

`x` Exactifying method.

`z` Zero method. This method moves all of the vertices to the first part. Its main use is to stop the bipartitioning process, if some condition is true, when mapping onto variable-sized architectures (see section 3.2.3).

8.3.4 Vertex partitioning (with overlap) strategy strings

Vertex partitioning is a special form of graph partitioning, in which graphs are partitioned into a prescribed number of parts by means of vertex separators rather than of edge separators like in Section 8.3.2. The load balance criterion also differs from common practice: the load to be balanced across all parts comprises not only the load of the vertices which belong to the part, but also the load of all the separator vertices which are their immediate neighbors. Consequently, the load of every separator vertex is accounted for several times, in each of the parts it separates.

Vertex partitioning strategies are made of methods, with optional parameters enclosed between curly braces, and separated by commas, in the form of *method*[{*parameters*}] . The currently available mapping methods are the following.

`e` K-way edge partitioning method. The parameters of the Fiduccia-Mattheyses method are listed below.

`strat=`*strat*
: K-way partitioning strategy to be performed. It is in fact a k-way mapping strategy, that is applied to a complete target graph of as many vertices as the prescribed number of parts. The syntax of mapping strategy strings is defined in Section 8.3.2.

`f` Fiduccia-Mattheyses method. The parameters of the Fiduccia-Mattheyses method are listed below.

`bal=`*rat*
: Set the maximum weight imbalance ratio to the given fraction of the subgraph vertex weight. Common values are around 0.01, that is, one percent.

`move=`*nbr*
: Maximum number of hill-climbing moves that can be performed before a pass ends. During each of its passes, the Fiduccia-Mattheyses algorithm repeatedly moves vertices between parts so as to minimize the cost function. A pass completes either when all of the vertices have been moved once, or if too many swaps that do not decrease the value of the cost function have been performed. Setting this value to zero turns the Fiduccia-Mattheyses algorithm into a gradient-like method, which may be used to quickly refine partitions during the uncoarsening phase of the multilevel method.

`pass=`*nbr*
: Set the maximum number of optimization passes performed by the algorithm. The Fiduccia-Mattheyses algorithm stops as soon as a pass has not yielded any improvement of the cost function, or when the maximum number of passes has been reached. Value -1 stands for an infinite number of passes, that is, as many as needed by the algorithm to converge.

`h`
: Greedy-graph-growing method. This is a k-way version of the original algorithm, in which parts are grown one after the other. Consequently, depending on graph topology, this method is likely to yield disconnected parts, with higher probability as the number of part increases. This method has only one parameter.

  `pass=`*nbr*
  : Set the number of runs performed by the algorithm.

`m`
: Multilevel method. The parameters of the multilevel method are listed below.

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

  `low=`*strat*
  : Set the strategy that is used to compute the partition of the coarsest 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 vertices than the minimum number of vertices allowed.

  `vert=`*nbr*
  : Set the threshold minimum number of vertices per part 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 vertices than the minimum number of vertices allowed.

`r`
: Recursive bipartitioning algorithm. The parameters of the recursive bipartitioning method are listed below.

`sep=`*strat*
: Apply vertex (node) separation strategy *strat* to each bipartitioning job. A node separation strategy is made of one or several node separation methods, which can be combined by means of strategy operators. Node separation strategies are described in Section 8.3.6.

8.3.5 Ordering strategy strings

Ordering strategies are available both for graphs and for meshes. An ordering strategy is made of one or several 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 number of vertices 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 vertex load (weight) of the current subgraph. Integer.

    `mdeg`
    : The maximum degree of the current subgraph. Integer.

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

method[{*parameters*}]
: Graph or mesh ordering method. Available ordering methods are listed below.

The currently available ordering methods are the following.

`b` Blocking method. This method does not perform ordering by itself, but is used as post-processing to cut into blocks of smaller sizes the separators or large blocks produced by other ordering methods. This is not useful in the context of direct solving methods, because the off-diagonal blocks created by the splitting of large diagonal blocks are likely to be filled at factoring time. However, in the context of incomplete solving methods such as ILU(k) [31], it can lead to a significant reduction of the required memory space and time, because it helps carving large triangular blocks. The parameters of the blocking method are described below.

`cmin`=*wght*
: Set the minimum span of the resulting sub-blocks, in terms of column weights. When the graph has no vertex weights (that is, all columns have weight 1), `cmin` represents the minimum number of colums to be included within each sub-block. For unweighted graphs, blocks larger than twice this minimum weight are cut into sub-blocks of equal sizes (within one), having a number of columns comprised between *wght* and 2*wght*. For weighted graphs, the algorithm performs in a best effort to achieve this goal.
The definition of *size* depends on the size of the graph to order. Large graphs cannot afford very small values, because the number of blocks becomes much too large and limits the acceleration of BLAS 3 routines, while large values do not help reducing enough the complexity of ILU(k) solving.

`strat`=*strat*
: Ordering strategy to be performed. After the ordering strategy is applied, the resulting separators tree is traversed and all of the column blocks that are larger than 2*size* are split into smaller column blocks, without changing the ordering that has been computed.

`c` Compression method [2]. The parameters of the compression method are listed below.

`rat`=*rat*
: Set the compression ratio over which graphs and meshes will not be compressed. Useful values range between 0.7 and 0.8.

`cpr`=*strat*
: Ordering strategy to use on the compressed graph or mesh if its size is below the compression ratio times the size of the original graph or mesh.

`unc`=*strat*
: Ordering strategy to use on the original graph or mesh if the size of the compressed graph or mesh were above the compression ratio times the size of the original graph or mesh.

`d` Block Halo Approximate Minimum Degree method [49]. The parameters of the Halo Approximate Minimum Degree method are listed below. The Block Halo Approximate Minimum Fill method, described below, is more efficient and should be preferred.

`cmin`=*wght*
: Minimum weight per column block. All column blocks of weight smaller

than *wght* are amalgamated to their parent column block in the elimination tree, provided that it does not violate the `cmax` constraint.

`cmax`=*wght*
: Maximum weight over which a column block will not amalgamate one of its descendents in the elimination tree. This parameter is mainly designed to provide an upper bound for block size in the context of BLAS3 computations ; else, a huge value should be provided.

`frat`=*rat*
: Fill-in ratio over which some column block will not amalgamate one of its descendents in the elimination tree. Typical values range from 0.05 to 0.10.

`f` Block Halo Approximate Minimum Fill method. The parameters of the Halo Approximate Minimum Fill method are listed below.

`cmin`=*wght*
: Minimum weight per column block. All column blocks of weight smaller than *wght* are amalgamated to their parent column block in the elimination tree, provided that it does not violate the `cmax` constraint.

`cmax`=*size*
: Maximum weight over which a column block will not amalgamate one of its descendents in the elimination tree. This parameter is mainly designed to provide an upper bound for block size in the context of BLAS3 computations ; else, a huge value should be provided.

`frat`=*rat*
: Fill-in ratio over which some column block will not amalgamate one of its descendents in the elimination tree. Typical values range from 0.05 to 0.10.

`g` Gibbs-Poole-Stockmeyer method. This method is used on separators to reduce the number and extent of extra-diagonal blocks. If the number of extra-diagonal blocks is not relevant, the `s` method should be preferred. This method has only one parameter.

`pass`=*nbr*
: Set the number of sweeps performed by the algorithm.

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

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

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

`sep`=*strat*
: Set the node separation strategy that is used on every leaf of the separators tree to make it grow. Node separation strategies are described below, in section 8.3.6.

`o` Disconnected subgraph detection method. This method is used at the global level to search for connected components, and run independently the provided graph ordering strategy on each of them.

`strat=`*strat*
Ordering strategy to apply to each of the connected components.

`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 ; else, the `g` method should be preferred.

`v` Mesh-to-graph method. Available only for mesh ordering strategies. From the mesh to which this method applies is derived a graph, such that a graph vertex is associated with every node of the mesh, and a clique is created between all vertices which represent nodes that belong to the same element. A graph ordering strategy is then applied to the derived graph, and this ordering is projected back to the nodes of the mesh. This method is here for evaluation purposes only, as mesh ordering methods are generally more efficient than their graph ordering counterpart.

`strat=`*strat*
Graph ordering strategy to apply to the associated graph.

8.3.6 Node separation strategy strings

A node separation strategy is made of one or several 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.

  `levl`
  : The level of the subgraph in the separators tree, starting from zero at the root of the tree. Integer.

  `proc`
  : The number of processors on which the current subgraph is distributed at this level of the separators tree. This variable is available only when calling from routines of the PT-Scotch parallel library. Integer.

  `rank`
  : The rank of the current processor among the group of processors on which the current subgraph is distributed at this level of the separators tree. This variable is available only when calling from routines of the PT-Scotch parallel library, for instance to decide which node separation strategy should be used on which processor in a multi-sequential approach. Integer.

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

The currently available vertex separation methods are the following.

`b` Band method. Available only for graph separation strategies. This method builds a band graph of given width around the current separator of the graph to which it is applied, and calls a graph separation strategy to refine the equivalent separator of the band graph. Then, the refined separator of the band graph is projected back to the current graph. This method, presented in [9], was created to reduce the cost of separator refinement algorithms in a multilevel context, but it improves partition quality too. The parameters of the band separation method are listed below.

`bnd=`*strat*
: Set the vertex separation strategy to be used on the band graph.

`org=`*strat*
: Set the fallback vertex separation strategy to be used on the original graph if the band graph strategy could not be used. The three cases which require the use of this fallback strategy are the following. First, if the separator of the original graph is empty, which makes it impossible to compute a band graph. Second, if any part of the band graph to be built is of the same size as the one of the original graph. Third, if the application of the `bnd` vertex separation method to the band graph leads to a situation where both anchor vertices are placed in the same part.

`width=`*val*
: Set the width of the band graph. All graph vertices that are at a distance less than or equal to *val* from any separator vertex are kept in the band graph.

`e` Edge-separation method. Available only for graph separation strategies. This method builds vertex separators from edge separators, by the method proposed by Pothen and Fang [51], which uses a variant of the Hopcroft and Karp algorithm due to Duff [10]. This method is expensive and most often yields poorer results than direct vertex-oriented methods such as the vertex vertex Greedy-graph-growing and the vertex Fiduccia-Mattheyses algorithms. The parameters of the edge-separation method are listed below.

`bal=`*val*
: Set the load imbalance tolerance to *val*, which is a floating-point ratio expressed with respect to the ideal load of the partitions.

`strat=`*strat*
: Set the graph bipartitioning strategy that is used to compute the edge bipartition. The syntax of bipartitioning strategy strings is defined in Section 8.3.3.

`width=`*type*
: Select the width of the vertex separators built from edge separators. When *type* is set to `f`, fat vertex separators are built, that hold all of the ends of the edges of the edge cut. When it is set to `t`, a thin vertex separator is built by removing as many vertices as possible from the fat separator.

`f` Vertex Fiduccia-Mattheyses method. The parameters of the vertex Fiduccia-Mattheyses method are listed below.

`bal=`*rat*
: Set the maximum weight imbalance ratio to the given fraction of the weight of all node vertices. Common values are around 0.01, that is, one percent.

`move=`*nbr*
: Maximum number of hill-climbing moves that can be performed before a pass ends. During each of its passes, the vertex Fiduccia-Mattheyses algorithm repeatedly moves vertices from the separator to any of the two parts, so as to minimize the size of the separator. A pass completes either when all of the vertices have been moved once, or if too many swaps that do not decrease the size of the separator have been performed.

`pass=`*nbr*
: Set the maximum number of optimization passes performed by the algorithm. The vertex Fiduccia-Mattheyses algorithm stops as soon as a pass has not yielded any reduction of the size of the separator, or when the maximum number of passes has been reached. Value -1 stands for an infinite number of passes, that is, as many as needed by the algorithm to converge.

`g` Gibbs-Poole-Stockmeyer method. Available only for graph separation strategies. This method has only one parameter.

`pass=`*nbr*
: Set the number of sweeps performed by the algorithm.

`h` Vertex greedy-graph-growing method. This method has only one parameter.

`pass=`*nbr*
: Set the number of runs performed by the algorithm.

`m` Vertex multilevel method. The parameters of the vertex multilevel method are listed below.

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

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

`rat=`*rat*
: Set the threshold maximum coarsening ratio over which graphs or meshes 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 or mesh has fewer node vertices than the minimum number of vertices allowed.

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

`t` Thinner method. Available only for graph separation strategies. This method quickly eliminates all useless vertices of the current separator. It searches the separator for vertices that have no neighbors in one of the two parts, and moves these vertices to the part they are connected to. This method may be used to refine separators during the uncoarsening phase of the multilevel method, and is faster than a vertex Fiduccia-Mattheyses algorithm with {`move=0`}.

`v` Mesh-to-graph method. Available only for mesh separation strategies. From the mesh to which this method applies is derived a graph, such that a graph vertex is associated with every node of the mesh, and a clique is created between all vertices which represent nodes that belong to the same element. A graph separation strategy is then applied to the derived graph, and the separator is projected back to the nodes of the mesh. This method is here for evaluation purposes only, as mesh separation methods are generally more efficient than their graph separation counterpart.

`strat=`*strat*
: Graph separation strategy to apply to the associated graph.

`w` Graph separator viewer. Available only for graph separation strategies. Every call to this method results in the creation, in the current subdirectory, of partial mapping files called "`vgraphseparatevw_output_`*nnnnnnnn*`.map`",

where "*nnnnnnnn*" are increasing decimal numbers, which contain the current state of the two parts and the separator. These mapping files can be used as input by the `gout` program to produce displays of the evolving shape of the current separator and parts. This is mostly a debugging feature, but it can also have an illustrative interest. While it is only available for graph separation strategies, mesh separation strategies can indirectly use it through the mesh-to-graph separation method.

`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.

8.4 Target architecture handling routines

8.4.1 `SCOTCH_archAlloc`

Synopsis

```
SCOTCH_Arch * SCOTCH_archAlloc (void)
```

Description

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

Return values

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

8.4.2 `SCOTCH_archExit`

Synopsis

```
void SCOTCH_archExit (SCOTCH_Arch *  archptr)
scotchfarchexit (doubleprecision (*)  archdat)
```

Description

The `SCOTCH_archExit` function frees the contents of a `SCOTCH_Arch` structure previously initialized by `SCOTCH_archInit`. All subsequent calls to `SCOTCH_arch` routines other than `SCOTCH_archInit`, using this structure as parameter, may yield unpredictable results.

8.4.3 SCOTCH_archInit

Synopsis

```
int SCOTCH_archInit (SCOTCH_Arch *  archptr)

scotchfarchinit (doubleprecision (*)  archdat,
                 integer              ierr)
```

Description

The SCOTCH_archInit function initializes a SCOTCH_Arch structure so as to make it suitable for future operations. It should be the first function to be called upon a SCOTCH_Arch structure. When the target architecture data is no longer of use, call function SCOTCH_archExit to free its internal structures.

Return values

SCOTCH_archInit returns 0 if the architecture structure has been successfully initialized, and 1 else.

8.4.4 SCOTCH_archLoad

Synopsis

```
int SCOTCH_archLoad (SCOTCH_Arch *  archptr,
                     FILE *         stream)

scotchfarchload (doubleprecision (*)  archdat,
                 integer              fildes,
                 integer              ierr)
```

Description

The SCOTCH_archLoad routine fills the SCOTCH_Arch structure pointed to by archptr with the source graph description available from stream stream in the Scotch target architecture format (see Section 6.4).

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 architecture file.

Return values

SCOTCH_archLoad returns 0 if the target architecture structure has been successfully allocated and filled with the data read, and 1 else.

8.4.5 SCOTCH_archName

Synopsis

```
const char * SCOTCH_archName (const SCOTCH_Arch *  archptr)
```

```
scotchfarchname (doubleprecision (*)  archdat,
                 character (*)        chartab,
                 integer              charnbr)
```

Description

The SCOTCH_archName function returns a string containing the name of the architecture pointed to by archptr. Since Fortran routines cannot return string pointers, the scotchfarchname routine takes as second and third parameters a character() array to be filled with the name of the architecture, and the integer size of the array, respectively. If the array is of sufficient size, a trailing nul character is appended to the string to materialize the end of the string (this is the C style of handling character strings).

Return values

SCOTCH_archName returns a non-null character pointer that points to a null-terminated string describing the type of the architecture.

8.4.6 SCOTCH_archSave

Synopsis

```
int SCOTCH_archSave (const SCOTCH_Arch *  archptr,
                     FILE *               stream)
```

```
scotchfarchsave (doubleprecision (*)  archdat,
                 integer              fildes,
                 integer              ierr)
```

Description

The SCOTCH_archSave routine saves the contents of the SCOTCH_Arch structure pointed to by archptr to stream stream, in the Scotch target architecture format (see section 6.4).

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 architecture file.

Return values

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

8.4.7 SCOTCH_archSize

Synopsis

```
SCOTCH_Num SCOTCH_archSize (const SCOTCH_Arch *  archptr)
```

```
scotchfarchsize (doubleprecision (*)  archdat,
                 integer*num          archnbr)
```

Description

The SCOTCH_archSize function returns the number of nodes of the given target architecture. The Fortran routine has a second parameter, of integer type, which is set on return with the number of nodes of the target architecture.

Return values

SCOTCH_archSize returns the number of nodes of the target architecture.

8.4.8 SCOTCH_archSizeof

Synopsis

```
int SCOTCH_archSizeof (void)
scotchfarchsizeof (integer  size )
```

Description

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

8.5 Target architecture creation routines

8.5.1 SCOTCH_archBuild0 / SCOTCH_archBuild

Synopsis

```
int SCOTCH_archBuild0 (SCOTCH_Arch *         archptr,
                       const SCOTCH_Graph *  grafptr,
                       const SCOTCH_Num      listnbr,
                       const SCOTCH_Num *    listtab,
                       const SCOTCH_Strat *  straptr)
int SCOTCH_archBuild (SCOTCH_Arch *         archptr,
                      const SCOTCH_Graph *  grafptr,
                      const SCOTCH_Num      listnbr,
                      const SCOTCH_Num *    listtab,
                      const SCOTCH_Strat *  straptr)
scotchfarchbuild0 (doubleprecision (*)  archdat,
                   doubleprecision (*)  grafdat,
                   integer*num          listnbr,
                   integer*num (*)      listtab,
                   doubleprecision (*)  stradat,
                   integer              ierr)
```

```
scotchfarchbuild (doubleprecision (*)  archdat,
                  doubleprecision (*)  grafdat,
                  integer*num          listnbr,
                  integer*num (*)      listtab,
                  doubleprecision (*)  stradat,
                  integer              ierr)
```

Description

The SCOTCH_archBuild0 routine fills the architecture structure pointed to by archptr with the "deco 1" (that is, a compiled form of a "deco 0") decomposition-defined target architecture computed by applying the graph bipartitioning strategy pointed to by straptr to the architecture graph pointed to by grafptr.

When listptr is not NULL and listnbr is greater than zero, the decomposition-defined architecture is restricted to the listnbr vertices whose indices are given in the array pointed to by listtab, from listtab [0] to listtab[listnbr - 1]. These indices should have the same base value as the one of the graph pointed to by grafptr, that is, be in the range from 0 to **vertnbr** − 1 if the graph base is 0, and from 1 to **vertnbr** if the graph base is 1.

Graph bipartitioning strategies are declared by means of the SCOTCH_strat GraphBipart function, described in page 154. The syntax of bipartitioning strategy strings is defined in section 8.3.2, page 66. Additional information may be obtained from the manual page of amk_grf, the stand-alone executable that builds decomposition-defined target architecture files from source graph files, available at page 38.

At the time being, SCOTCH_archBuild is equivalent to SCOTCH_arch Build0. In future releases, it is planned that SCOTCH_archBuild will either behave as SCOTCH_archBuild0 or SCOTCH_archBuild2, depending on target graph size. For target graphs of small sizes, users are invited to use explicitly the SCOTCH_archBuild0 routine.

Return values

SCOTCH_archBuild0 returns 0 if the decomposition-defined architecture has been successfully computed, and 1 else.

8.5.2 SCOTCH_archBuild2

Synopsis

```
int SCOTCH_archBuild2 (SCOTCH_Arch *         archptr,
                       const SCOTCH_Graph *  grafptr,
                       const SCOTCH_Num      listnbr,
                       const SCOTCH_Num *    listtab)

scotchfarchbuild2 (doubleprecision (*)  archdat,
                   doubleprecision (*)  grafdat,
                   integer*num          listnbr,
                   integer*num (*)      listtab,
                   integer              ierr)
```

Description

The SCOTCH_archBuild2 routine fills the architecture structure pointed to by archptr with the "deco 2" decomposition-defined target architecture corresponding to the graph pointed to by grafptr. Since the computation of the decomposition is performed by means of graph coarsening, unlike SCOTCH_archBuild, no bipartitioning strategy has to be provided.

When listptr is not NULL and listnbr is greater than zero, the decomposition-defined architecture is restricted to the listnbr vertices whose indices are given in the array pointed to by listtab, from listtab[0] to listtab[listnbr - 1]. These indices should have the same base value as that of the graph pointed to by grafptr, that is, be in the range from 0 to **vertnbr** − 1 if the graph base is 0, and from 1 to **vertnbr** if the graph base is 1.

Additional information may be obtained from the manual page of amk_grf, the stand-alone executable that builds decomposition-defined target architecture files from source graph files, available at page 38.

Return values

SCOTCH_archBuild returns 0 if the decomposition-defined architecture has been successfully computed, and 1 else.

8.5.3 SCOTCH_archCmplt

Synopsis

```
int SCOTCH_archCmplt (SCOTCH_Arch *     archptr,
                      const SCOTCH_Num  vertnbr)

scotchfarchcmplt (doubleprecision (*)  archdat,
                  integer*num          vertnbr,
                  integer              ierr)
```

Description

The SCOTCH_archCmplt routine fills the SCOTCH_Arch structure pointed to by archptr with the description of a complete graph architecture with vertnbr processors, which can be used as input to SCOTCH_graphMap to perform graph partitioning. A shortcut to this is to use the SCOTCH_graphPart routine.

Return values

SCOTCH_archCmplt returns 0 if the complete graph target architecture has been successfully built, and 1 else.

8.5.4 SCOTCH_archCmpltw

Synopsis

```
int SCOTCH_archCmpltw (SCOTCH_Arch *     archptr,
                       const SCOTCH_Num  vertnbr,
                       const SCOTCH_Num  velotab)

scotchfarchcmplt (doubleprecision (*)  archdat,
                  integer*num          vertnbr,
                  integer*num (*)      velotab,
                  integer              ierr)
```

Description

The SCOTCH_archCmpltw routine fills the SCOTCH_Arch structure pointed to by archptr with the description of a weighted complete graph architecture with vertnbr processors. The relative weights of the processors are given in the velotab array. Once the target architecture has been created, it can be used as input to SCOTCH_graphMap to perform weighted graph partitioning.

Return values

SCOTCH_archCmpltw returns 0 if the weighted complete graph target architecture has been successfully built, and 1 else.

8.5.5 SCOTCH_archHcub

Synopsis

```
int SCOTCH_archHcub (SCOTCH_Arch *     archptr,
                     const SCOTCH_Num  hdimval)

scotchfarchhcub (doubleprecision (*)  archdat,
                 integer*num          hdimval,
                 integer              ierr)
```

Description

The SCOTCH_archHcub routine fills the SCOTCH_Arch structure pointed to by archptr with the description of a hypercube graph architecture of dimension hdimval.

Return values

SCOTCH_archHcub returns 0 if the hypercube target architecture has been successfully built, and 1 else.

8.5.6 SCOTCH_archLtleaf

Synopsis

```
int SCOTCH_archLtleaf (SCOTCH_Arch *       archptr,
                       const SCOTCH_Num    levlnbr,
                       const SCOTCH_Num *  sizetab,
                       const SCOTCH_Num *  linktab,
                       const SCOTCH_Num    permnbr,
                       const SCOTCH_Num *  permtab)

scotchfarchltleaf (doubleprecision (*)  archdat,
                   integer*num          levlnbr,
                   integer*num (*)      sizetab,
                   integer*num (*)      linktab,
                   integer*num          permnbr,
                   integer*num (*)      permtab,
                   integer              ierr)
```

Description

The SCOTCH_archLtleaf routine fills the SCOTCH_Arch structure pointed to by archptr with the description of a labeled, tree-shaped, hierarchical graph architecture with $\sum_{i=0}^{\texttt{levlnbr}-1} \texttt{sizetab}[i]$ processors. Level 0 is the root of the tree. For each level i, with $0 \leq i < \texttt{levlnbr}$, sizetab[$i$] is the number of childs at level $(i+1)$ of each node at level i, and linktab[i] is the cost of communication between processors the first common ancestor of which belongs to this level. See Section 6.4.2, page 29, for an example of this architecture.

Return values

SCOTCH_archLtleaf returns 0 if the labeled tree-leaf target architecture has been successfully built, and 1 else.

8.5.7 SCOTCH_archMesh2

Synopsis

```
int SCOTCH_archMesh2 (SCOTCH_Arch *     archptr,
                      const SCOTCH_Num  xdimval,
                      const SCOTCH_Num  ydimval)

scotchfarchmesh2 (doubleprecision (*)  archdat,
                  integer*num          xdimval,
                  integer*num          ydimval,
                  integer              ierr)
```

Description

The SCOTCH_archMesh2 routine fills the SCOTCH_Arch structure pointed to by archptr with the description of a 2D mesh architecture with xdimval $\times$ ydimval processors.

Return values

SCOTCH_archMesh2 returns 0 if the 2D mesh target architecture has been successfully built, and 1 else.

8.5.8 SCOTCH_archMesh3

Synopsis

```
int SCOTCH_archMesh3 (SCOTCH_Arch *     archptr,
                      const SCOTCH_Num  xdimval,
                      const SCOTCH_Num  ydimval,
                      const SCOTCH_Num  zdimval)

scotchfarchmesh3 (doubleprecision (*)  archdat,
                  integer*num          xdimval,
                  integer*num          ydimval,
                  integer*num          zdimval,
                  integer              ierr)
```

Description

The SCOTCH_archMesh3 routine fills the SCOTCH_Arch structure pointed to by archptr with the description of a 3D mesh architecture with **xdimval** $\times$ **ydimval** $\times$ **zdimval** processors.

Return values

SCOTCH_archMesh3 returns 0 if the 3D mesh target architecture has been successfully built, and 1 else.

8.5.9 SCOTCH_archMeshX

Synopsis

```
int SCOTCH_archMeshX (SCOTCH_Arch *       archptr,
                      const SCOTCH_Num    dimnnbr,
                      const SCOTCH_Num *  dimntab)

scotchfarchmeshx (doubleprecision (*)  archdat,
                  integer*num          dimnnbr,
                  integer*num          dimntab,
                  integer              ierr)
```

Description

The SCOTCH_archMeshX routine fills the SCOTCH_Arch structure pointed to by archptr with the description of a dimnnbr-dimension mesh architecture with $\prod_d$ **dimntab**$[d]$ processors. The maximum number of dimensions is defined at compile-time.

Return values

SCOTCH_archMeshX returns 0 if the dimnnbr-dimension mesh target architecture has been successfully built, and 1 else.

8.5.10 SCOTCH_archSub

Synopsis

```
int SCOTCH_archSub (SCOTCH_Arch *       subarchptr,
                    SCOTCH_Arch *       orgarchptr,
                    const SCOTCH_Num    vnumnbr,
                    const SCOTCH_Num *  vnumtab)

scotchfarchsub (doubleprecision (*)  subarchdat,
                doubleprecision (*)  orgarchdat,
                integer*num          vnumnbr,
                integer*num          vnumtab,
                integer              ierr)
```

Description

The SCOTCH_archSub routine fills the SCOTCH_Arch structure pointed to by subarchptr with the description of a subset of the orgarchptr architecture, restricted to vertnbr processors which are listed in the vnumtab array. The order in which the processor indices in the original architecture are stored in the vnumtab array defines the rank of these processors in the sub-architecture.

Since the sub-architecture depends on the original architecture, the latter must not be de-allocated (by way of SCOTCH_archExit) as long as the sub-architecture is being used.

Return values

SCOTCH_archSub returns 0 if the target sub-architecture has been successfully built, and 1 else.

8.5.11 SCOTCH_archTleaf

Synopsis

```
int SCOTCH_archTleaf (SCOTCH_Arch *       archptr,
                      const SCOTCH_Num    levlnbr,
                      const SCOTCH_Num *  sizetab,
                      const SCOTCH_Num *  linktab)

scotchfarchtleaf (doubleprecision (*)  archdat,
                  integer*num          levlnbr,
                  integer*num (*)      sizetab,
                  integer*num (*)      linktab,
                  integer              ierr)
```

Description

The SCOTCH_archTleaf routine fills the SCOTCH_Arch structure pointed to by archptr with the description of a tree-shaped, hierarchical graph architecture with $\sum_{i=0}^{\texttt{levlnbr}-1} \texttt{sizetab}[i]$ processors. Level 0 is the root of the tree.

For each level i, with $0 \leq i <$ `levlnbr`, `sizetab[`i`]` is the number of childs at level $(i+1)$ of each node at level i, and `linktab[`i`]` is the cost of communication between processors the first common ancestor of which belongs to this level. See Section 6.4.2, page 28, for an example of this architecture.

Return values

`SCOTCH_archTleaf` returns 0 if the tree-leaf target architecture has been successfully built, and 1 else.

8.5.12 `SCOTCH_archTorus2`

Synopsis

```
int SCOTCH_archTorus2 (SCOTCH_Arch *     archptr,
                       const SCOTCH_Num  xdimval,
                       const SCOTCH_Num  ydimval)

scotchfarchtorus2 (doubleprecision (*)  archdat,
                   integer*num          xdimval,
                   integer*num          ydimval,
                   integer              ierr)
```

Description

The `SCOTCH_archTorus2` routine fills the `SCOTCH_Arch` structure pointed to by `archptr` with the description of a 2D torus architecture with `xdimval` $\times$ `ydimval` processors.

Return values

`SCOTCH_archTorus2` returns 0 if the 2D torus target architecture has been successfully built, and 1 else.

8.5.13 `SCOTCH_archTorus3`

Synopsis

```
int SCOTCH_archTorus3 (SCOTCH_Arch *     archptr,
                       const SCOTCH_Num  xdimval,
                       const SCOTCH_Num  ydimval,
                       const SCOTCH_Num  zdimval)

scotchfarchtorus3 (doubleprecision (*)  archdat,
                   integer*num          xdimval,
                   integer*num          ydimval,
                   integer*num          zdimval,
                   integer              ierr)
```

Description

The SCOTCH_archTorus3 routine fills the SCOTCH_Arch structure pointed to by archptr with the description of a 3D torus architecture with **xdimval** × ydimval × **zdimval** processors.

Return values

SCOTCH_archTorus3 returns 0 if the 3D torus target architecture has been successfully built, and 1 else.

8.5.14 SCOTCH_archTorusX

Synopsis

```
int SCOTCH_archTorusX (SCOTCH_Arch *       archptr,
                       const SCOTCH_Num    dimnnbr,
                       const SCOTCH_Num *  dimntab)

scotchfarchtorusx (doubleprecision (*)  archdat,
                   integer*num          dimnnbr,
                   integer*num          dimntab,
                   integer              ierr)
```

Description

The SCOTCH_archTorusX routine fills the SCOTCH_Arch structure pointed to by archptr with the description of a dimnnbr-dimension torus architecture with $\prod_d$ **dimntab**$[d]$ processors. The maximum number of dimensions is defined at compile-time.

Return values

SCOTCH_archTorusX returns 0 if the dimnnbr-dimension mesh target architecture has been successfully built, and 1 else.

8.5.15 SCOTCH_archVcmplt

Synopsis

```
int SCOTCH_archVcmplt (SCOTCH_Arch *  archptr)

scotchfarchvcmplt (doubleprecision (*)  archdat,
                   integer              ierr)
```

Description

The SCOTCH_archVcmplt routine fills the SCOTCH_Arch structure pointed to by archptr with the description of a "variable-sized" complete graph architecture, which can be used as input to SCOTCH_graphMap to perform graph clustering (see Section 3.2.3).

Every domain of a variable-size architecture can always be bipartitioned into two subdomains. Consequently, when used in the context of a recursive bipartitioning algorithm, the algorithm will perform recursively until there is only

a single source graph vertex in some target domain, or some bipartitioning method assigns all its source graph vertices to one of the subdomains.

Return values

SCOTCH_archVcmplt returns 0 if the variable-sized complete graph target architecture has been successfully built, and 1 else.

8.5.16 SCOTCH_archVhcub

Synopsis

```
int SCOTCH_archVhcub (SCOTCH_Arch *  archptr)

scotchfarchvhcub (doubleprecision (*)  archdat,
                  integer              ierr)
```

Description

The SCOTCH_archVhcub routine fills the SCOTCH_Arch structure pointed to by archptr with the description of a "variable-sized" hypercube architecture, which can be used as input to SCOTCH_graphMap to perform graph clustering (see Section 3.2.3).

Every domain of a variable-size architecture can always be bipartitioned into two subdomains. Consequently, when used in the context of a recursive bipartitioning algorithm, the algorithm will perform recursively until there is only a single source graph vertex in some target domain, or some bipartitioning method assigns all its source graph vertices to one of the subdomains.

The difference of the variable-sized hypercube architecture with respect to the variable-sized complete graph architecture is that the cost of previously cut edges increases with the dimension of the hypercube. Hence, when some vertices whose edges have been cut previously, are placed in some part, their cut neighbors will tend to be put in a same part as well, on the other branch of the recursive bipartitioning tree, therefore increasing cluster locality.

Return values

SCOTCH_archVhcub returns 0 if the variable-sized hypercube target architecture has been successfully built, and 1 else.

8.6 Target domain handling routines

8.6.1 SCOTCH_archDomAlloc

Synopsis

```
SCOTCH_ArchDom * SCOTCH_archDomAlloc (void)
```

Description

The SCOTCH_archDomAlloc function allocates a memory area of a size sufficient to store a SCOTCH_ArchDom structure. It is the user's responsibility to free this memory when it is no longer needed, using the SCOTCH_memFree routine.

Return values

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

8.6.2 SCOTCH_archDomBipart

Synopsis

```
int SCOTCH_archDomBipart (SCOTCH_Arch *           archptr,
                          const SCOTCH_ArchDom *  domnptr,
                          SCOTCH_ArchDom *        dom0ptr,
                          SCOTCH_ArchDom *        dom1ptr)

scotchfarchdombipart (doubleprecision (*)  archdat,
                      doubleprecision (*)  domndat,
                      doubleprecision (*)  dom0dat,
                      doubleprecision (*)  dom1dat,
                      integer              ierr)
```

Description

The SCOTCH_archDomBipart function tries to split the domain referred to by domnptr into two disjoint subdomains referred to by dom0ptr and dom1ptr, in the target architecture referred to archptr.

Return values

SCOTCH_archDomBipart returns 0 if the domain could be bipartitioned, 1 if bipartitioning could not be performed (because the domain is terminal), and 2 on error.

8.6.3 SCOTCH_archDomFrst

Synopsis

```
int SCOTCH_archDomFrst (SCOTCH_Arch *     archptr,
                        SCOTCH_ArchDom *  domnptr)

scotchfarchdomfrst (doubleprecision (*)  archdat,
                    doubleprecision (*)  domndat,
                    integer              ierr)
```

Description

The SCOTCH_archDomFrst function initializes the domain structure referred to by domnptr with the biggest domain of the target architecture referred to archptr, that is, the domain that contains all terminal domains.

Return values

SCOTCH_archDomFrst returns 0 if the domain could be created, and 1 on error.

8.6.4 SCOTCH_archDomSize

Synopsis

```
SCOTCH_Num SCOTCH_archDomSize (SCOTCH_Arch *           archptr,
                               const SCOTCH_ArchDom *  domnptr)

scotchfarchdomsize (doubleprecision (*)  archdat,
                    doubleprecision (*)  domndat,
                    integer*num          sizeval)
```

Description

The SCOTCH_archDomSize function returns the size of the domain referred to by domnptr, that is, the number of terminal domains comprised in the domain, within the architecture referred to by archptr. The Fortran routine has a third parameter, of SCOTCH_Num type, which is set on return with the domain size.

Return values

SCOTCH_archDomSize yields an integer value of type SCOTCH_Num that ranges between 1 and the number of terminal domains in the architecture.

8.6.5 SCOTCH_archDomSizeof

Synopsis

```
int SCOTCH_archDomSizeof (void)

scotchfarchdomsizeof (integer  size )
```

Description

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

8.6.6 SCOTCH_archDomTerm

Synopsis

```
int SCOTCH_archDomTerm (SCOTCH_Arch *     archptr,
                        SCOTCH_ArchDom *  domnptr,
                        const SCOTCH_Num  domnnum)
```

```
scotchfarchdomterm (doubleprecision (*)  archdat,
                    doubleprecision (*)  domndat,
                    integer*num          domnnum,
                    integer              ierr)
```

Description

The SCOTCH_archDomTerm function initializes the domain structure referred to by domnptr to correspond to the terminal domain of index domnnum in the target architecture referred to archptr.

Applying the archDomNum function to this domain yields back domnnum.

Return values

SCOTCH_archDomTerm returns 0 if the domain could be created, 1 if domnnum does not correspond to a valid terminal number for this architecture, and 2 on error.

8.6.7 SCOTCH_archDomWght

Synopsis

```
SCOTCH_Num SCOTCH_archDomWght (SCOTCH_Arch *           archptr,
                               const SCOTCH_ArchDom *  domnptr)

scotchfarchdomwght (doubleprecision (*)  archdat,
                    doubleprecision (*)  domndat,
                    integer*num          wghtval)
```

Description

The SCOTCH_archDomWght function returns the weight of the domain referred to by domnptr in the architecture referred to by archptr. The Fortran routine has a third parameter, of SCOTCH_Num type, which is set on return with this weight. The weight of a domain is the sum of the weights of all the terminal domains included within this domain.

Return values

SCOTCH_archDomWght yields an integer value of type SCOTCH_Num that ranges between 1 and the sum of the weights of all terminal domains in the architecture.

8.6.8 SCOTCH_archDomDist

Synopsis

```
SCOTCH_Num SCOTCH_archDomDist (SCOTCH_Arch *           archptr,
                               const SCOTCH_ArchDom *  dom0ptr,
                               const SCOTCH_ArchDom *  dom1ptr)
```

```
scotchfarchdomwght (doubleprecision (*)  archdat,
                    doubleprecision (*)  dom0dat,
                    doubleprecision (*)  dom1dat,
                    integer*num          distval)
```

Description

The SCOTCH_archDomDist function returns the estimated distance between the two domains referred to by dom0ptr and dom1ptr in the architecture referred to by archptr. The Fortran routine has a fourth parameter, of SCOTCH_Num type, which is set on return with this distance.

The desirable properties of distance functions are described in Section 3.2. Basically, they should provide more accurate results as domain sizes decrease and as distance decreases.

Return values

SCOTCH_archDomDist yields an integer value of type SCOTCH_Num that is always greater than or equal to zero. It is equal to the maximum available integer value when the two domains belong to different connected components of a disconnected target architecture.

8.6.9 SCOTCH_archDomNum

Synopsis

```
SCOTCH_Num SCOTCH_archDomNum (SCOTCH_Arch *           archptr,
                              const SCOTCH_ArchDom *  domnptr)

scotchfarchdomnum (doubleprecision (*)  archdat,
                   doubleprecision (*)  domndat,
                   integer*num          domnnum)
```

Description

The SCOTCH_archDomNum function returns the smallest number of terminal domain included within the domain referred to by domnptr of the architecture referred to by archptr. The Fortran routine has a third parameter, of SCOTCH_Num type, which is set on return with this terminal number.

Return values

SCOTCH_archDomNum yields an integer value of type SCOTCH_Num that ranges between 0 and the number of terminal domains in the architecture, minus 1.

8.7 Graph handling routines

8.7.1 SCOTCH_graphAlloc

Synopsis

```
SCOTCH_Graph * SCOTCH_graphAlloc (void)
```

Description

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

Return values

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

8.7.2 SCOTCH_graphBase

Synopsis

```
int SCOTCH_graphBase (SCOTCH_Graph *  grafptr,
                      SCOTCH_Num      baseval)

scotchfgraphbase (doubleprecision (*)  grafdat,
                  integer*num          baseval,
                  integer*num          oldbaseval)
```

Description

The SCOTCH_graphBase routine sets the base of all graph indices according to the given base value, and returns the old base value. This routine is a helper for applications that do not handle base values properly.

In Fortan, the old base value is returned in the third parameter of the function call.

Return values

SCOTCH_graphBase returns the old base value.

8.7.3 SCOTCH_graphBuild

Synopsis

```
int SCOTCH_graphBuild (SCOTCH_Graph *      grafptr,
                       const SCOTCH_Num    baseval,
                       const SCOTCH_Num    vertnbr,
                       const SCOTCH_Num *  verttab,
                       const SCOTCH_Num *  vendtab,
                       const SCOTCH_Num *  velotab,
                       const SCOTCH_Num *  vlbltab,
                       const SCOTCH_Num    edgenbr,
                       const SCOTCH_Num *  edgetab,
                       const SCOTCH_Num *  edlotab)
```

```
scotchfgraphbuild (doubleprecision (*)  grafdat,
                   integer*num          baseval,
                   integer*num          vertnbr,
                   integer*num (*)      verttab,
                   integer*num (*)      vendtab,
                   integer*num (*)      velotab,
                   integer*num (*)      vlbltab,
                   integer*num          edgenbr,
                   integer*num (*)      edgetab,
                   integer*num (*)      edlotab,
                   integer              ierr)
```

Description

The SCOTCH_graphBuild routine fills the 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). vertnbr is the number of vertices. verttab is the adjacency index array, of size (**vertnbr** + 1) if the edge array is compact (that is, if vendtab equals **verttab** + 1 or NULL), or of size vertnbr else. vendtab is the adjacency end index array, of size vertnbr if it is disjoint from verttab. velotab is the vertex load array, of size vertnbr if it exists. vlbltab is the vertex label array, of size vertnbr if it exists. edgenbr is the number of arcs (that is, twice the number of edges). edgetab is the adjacency array, of size at least edgenbr (it can be more if the edge array is not compact). edlotab is the arc load array, of size edgenbr if it exists.

The vendtab, velotab, vlbltab and edlotab arrays are optional, and a NULL pointer can be passed as argument whenever they are not defined. Since, in Fortran, there is no null reference, passing the scotchfgraphbuild routine a reference equal to verttab in the velotab or vlbltab fields makes them be considered as missing arrays. The same holds for edlotab when it is passed a reference equal to edgetab. Setting vendtab to refer to one cell after verttab yields the same result, as it is the exact semantics of a compact vertex array.

To limit memory consumption, SCOTCH_graphBuild does not copy array data, but instead references them in the SCOTCH_Graph structure. Therefore, great care should be taken not to modify the contents of the arrays passed to SCOTCH_graphBuild as long as the graph structure is in use. Every update of the arrays should be preceded by a call to SCOTCH_graphFree, to free internal graph structures, and eventually followed by a new call to SCOTCH_graphBuild to re-build these internal structures so as to be able to use the new 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, to call the SCOTCH_graphCheck routine on the newly created SCOTCH_Graph structure before calling any other LIBSCOTCH routine.

Return values

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

8.7.4 SCOTCH_graphCheck

Synopsis

```
int SCOTCH_graphCheck (const SCOTCH_Graph *  grafptr)

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

Description

The SCOTCH_graphCheck routine checks the consistency of the given SCOTCH_Graph structure. It can be used in client applications to determine if a graph that has been created from used-generated data by means of the SCOTCH_graphBuild routine is consistent, prior to calling any other routines of the libScotch library.

Return values

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

8.7.5 SCOTCH_graphCoarsen

Synopsis

```
int SCOTCH_graphCoarsen (const SCOTCH_Graph *  finegrafptr,
                         const SCOTCH_Num      coarvertnbr,
                         const double          coarrat,
                         const SCOTCH_Num      flagval,
                         SCOTCH_Graph *        coargrafptr,
                         SCOTCH_Num *          coarmulttab)

scotchfgraphcoarsen (doubleprecision (*)  finegrafdat,
                     integer*num          coarvertnbr,
                     doubleprecision      coarrat,
                     integer*num          flagval,
                     doubleprecision (*)  coargrafdat,
                     integer*num (*)      coarmulttab,
                     integer              ierr)
```

Description

The SCOTCH_graphCoarsen routine creates, in the SCOTCH_Graph structure coargrafdat pointed to by coargrafptr, a graph coarsened from the SCOTCH_Graph structure finegrafdat pointed to by finegrafptr. The coarsened graph is created only if it comprises more than coarvert nbr 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. When `SCOTCH_COARSEN NOMERGE` 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.

The `coarmulttab` array should be of a size big enough to store multinode data for the resulting coarsened graph. Hence, the size of the array must be at least twice the maximum expected number of local coarse vertices, according to the prescribed coarsening ratio `coarrat`. Upon successful completion, this array will contain pairs of consecutive `SCOTCH_Num` values, representing the indices of the two fine vertices that have been coarsened into each of the coarse vertices. When a vertex has been coarsened with itself, its two multinode values are identical.

`coargrafdat` must have been initialized with the `SCOTCH_graphInit` routine before `SCOTCH_graphCoarsen` is called.

Return values

`SCOTCH_graphCoarsen` 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.

8.7.6 `SCOTCH_graphCoarsenBuild`

Synopsis

```
int SCOTCH_graphCoarsenBuild (SCOTCH_Graph *    finegrafptr,
                              const SCOTCH_Num  coarvertnbr,
                              SCOTCH_Num *      finematetab,
                              SCOTCH_Graph *    coargrafptr,
                              SCOTCH_Num *      coarmulttab)

scotchfgraphcoarsenbuild (doubleprecision (*)  finegrafdat,
                          integer*num          coarvertnbr,
                          integer*num (*)      finematetab,
                          doubleprecision (*)  coargrafdat,
                          integer*num (*)      coarmulttab,
                          integer              ierr)
```

Description

The `SCOTCH_graphCoarsenBuild` routine creates, in the `SCOTCH_Graph` structure `coargrafdat` pointed to by `coargrafptr`, a graph with `coar vertnbr` vertices, coarsened from the `SCOTCH_Graph` structure `finegraf dat` pointed to by `finegrafptr`, using the matching provided by `finemate tab`.

On input, the `finematetab` mating array should contain the indices of the mates chosen for each vertex of the fine graph. When some vertex is mated to itself, its array cell value is equal to its own index. Upon successful completion,

this array is updated so as to contain fine-to-coarse indices: each array cell contains the index of the coarse vertex created from the given fine vertex.

The finematetab mating array and its associated number of coarse vertices coarvertnbr may have been computed using the SCOTCH_graphCoarsenMatch routine. Indeed, calling the SCOTCH_graphCoarsenMatch and SCOTCH_graphCoarsenBuild routines in sequence amounts to calling the SCOTCH_graphCoarsen routine, yet additionally publicizing the finematetab array.

The coarmulttab array should be of a size big enough to store multinode data for the resulting coarsened graph, that is, twice the value of coarvertnbr. Upon successful completion, this array will contain pairs of consecutive SCOTCH_Num values, representing the indices of the two fine vertices that have been coarsened into each of the coarse vertices. When a vertex has been coarsened with itself, the two multinode values are identical.

coargrafdat must have been initialized with the SCOTCH_graphInit routine before SCOTCH_graphCoarsenBuild is called.

Return values

SCOTCH_graphCoarsenBuild returns 0 if the coarse graph structure has been successfully created, and 1 on error.

8.7.7 SCOTCH_graphCoarsenMatch

Synopsis

```
int SCOTCH_graphCoarsenMatch (SCOTCH_Graph *    finegrafptr,
                              SCOTCH_Num *      coarvertptr,
                              const double      coarrat,
                              const SCOTCH_Num  flagval,
                              SCOTCH_Num *      finematetab)

scotchfgraphcoarsenmatch (doubleprecision (*)  finegrafdat,
                          integer*num          coarvertnbr,
                          doubleprecision      coarrat,
                          integer*num          flagval,
                          integer*num (*)      finematetab,
                          integer              ierr)
```

Description

The SCOTCH_graphCoarsenMatch routine fills the finematetab array with a matching of the vertices of the SCOTCH_Graph structure finegrafdat pointed to by finegrafptr. The matching is computed only if it amounts to the creation of more than coarvertnbr (that is, the value pointed to by coarvertptr in the C interface) coarse 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 matching. When SCOTCH_COARSENNOMERGE is set, isolated vertices are never matched with other vertices. This

preserves the topology of the graph, at the expense of a higher coarsening ratio.

The finematetab array must be of a size sufficient to hold as many SCOTCH_Num values as the number of vertices in the finegrafdat graph. Upon successful completion, this array will contain the indices of the mates chosen for each vertex of the provided graph. When some vertex is mated to itself, its array cell value is equal to its own index. Additionally, coarvertnbr will be set to the number of coarse vertices associated with the matching. This number is equal to the number of vertices in the provided graph, minus the number of matched pairs of vertices, since in a subsequent coarsening process, each pair should see its two matched vertices collapsed into a single coarse vertex.

The mating array and its associated number of coarse vertices can be used by the SCOTCH_graphCoarsenBuild routine. Indeed, calling the SCOTCH_graphCoarsenMatch and SCOTCH_graphCoarsenBuild routines in sequence amounts to calling the SCOTCH_graphCoarsen routine, yet additionally publicizing the finematetab array.

Return values

SCOTCH_graphCoarsenMatch returns 0 if a matching has been successfully computed, 1 if the matching was not computed because it did not enforce the threshold parameters, and 2 on error.

8.7.8 SCOTCH_graphColor

Synopsis

```
int SCOTCH_graphColor (const SCOTCH_Graph *  grafptr,
                       SCOTCH_Num *          colotab,
                       SCOTCH_Num *          coloptr,
                       SCOTCH_Num            flagval)

scotchfgraphcolor (doubleprecision (*)  grafdat,
                   integer*num (*)      colotab,
                   integernum           colonbr,
                   integernum           flagval,
                   integer              ierr)
```

Description

The SCOTCH_graphColor routine computes a coloring of the graph vertices. The colotab array is filled with color values, and the number of colors found is placed into the integer variable colonbr, pointed to by coloptr.

The computed coloring is not guaranteed to be maximal. Indeed, the only algorithm currently implemented is a variant of Luby's algorithm. Due to the operations of this algorithm, the first colors are likely to have many more representatives than the last colors.

Like for partition arrays, color values are *not* based: color values range from 0 to (colonbr − 1).

The flag value `flagval` is currently not used. It may be used in the future to select a coloring method. At the time being, a value of 0 should be provided.

Return values

`SCOTCH_graphColor` returns 0 if the graph coloring has been successfully computed, and 1 else.

8.7.9 `SCOTCH_graphData`

Synopsis

```
void SCOTCH_graphData (const SCOTCH_Graph *  grafptr,
                       SCOTCH_Num *          baseptr,
                       SCOTCH_Num *          vertptr,
                       SCOTCH_Num **         verttab,
                       SCOTCH_Num **         vendtab,
                       SCOTCH_Num **         velotab,
                       SCOTCH_Num **         vlbltab,
                       SCOTCH_Num *          edgeptr,
                       SCOTCH_Num **         edgetab,
                       SCOTCH_Num **         edlotab)

scotchfgraphdata (doubleprecision (*)  grafdat,
                  integer*num (*)      indxtab,
                  integer*num          baseval,
                  integer*num          vertnbr,
                  integer*idx          vertidx,
                  integer*idx          vendidx,
                  integer*idx          veloidx,
                  integer*idx          vlblidx,
                  integer*num          edgenbr,
                  integer*idx          edgeidx,
                  integer*num          edloidx)
```

Description

The `SCOTCH_graphData` routine is the dual of the `SCOTCH_graphBuild` 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). `vertptr` is the pointer to a location that will hold the number of vertices. `verttab` is the pointer to a location that will hold the reference to the adjacency index array, of size `*vertptr` + 1 if the adjacency array is compact, or of size `*vertptr` else. `vendtab` is the pointer to a location that will hold the reference to the adjacency end index array, and is equal to `verttab` + 1 if the adjacency array is compact. `velotab` is the pointer to a location that will hold the reference to the vertex load array, of size `*vertptr`. `vlbltab` is the pointer to a location that will hold the reference to the vertex label array, of size `vertnbr`. `edgeptr` is the pointer

to a location that will hold the number of arcs (that is, twice the number of edges). edgetab is the pointer to a location that will hold the reference to the adjacency array, of size at least *edgeptr. edlotab is the pointer to a location that will hold the reference to the arc load array, of size *edgeptr.

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 scotchfgraphdata 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 vertidx, the index of verttab, if they do not exist. For instance, if some base array myarray (1) is passed as parameter indxtab, then the first cell of array verttab will be accessible as myarray(vertidx). 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 one 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. 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 8.1.5 for more information on this issue.

8.7.10 SCOTCH_graphDiamPV

Synopsis

```
SCOTCH_Num SCOTCH_graphDiamPV (const SCOTCH_Graph *  grafptr)

scotchfgraphdiampv (doubleprecision (*)  grafdat,
                    integernum           diamval)
```

Description

The SCOTCH_graphDiamPV routine computes the edge-weighted (pseudo-)diameter value of the given graph.

To do so, it selects a random vertex, computes the set of vertices at maximum distance from this vertex by means of Dijkstra's algorithm, selects a vertex from this set, and repeats the process as long as this maximum distance value increases. If the graph is not edge-weighted, neighboring vertices are assumed to be at distance 1 from each other; else, edge weights represent distances between vertices.

Return values

SCOTCH_graphDiamPV returns a positive value if the graph diameter has been successfully computed, the SCOTCH_NUMMAX maximum positive value if the graph is disconnected, and -1 on error.

8.7.11 SCOTCH_graphDump

Synopsis

```
int SCOTCH_graphDump (const SCOTCH_Graph *  grafptr,
                      const char *          prefptr,
                      const char *          suffptr,
                      FILE *                stream)
```

Description

The SCOTCH_graphDump routine outputs the contents of the SCOTCH_Graph structure pointed to by grafptr to stream stream, in the form of a C source code. The names of the data arrays that encode the various graph arrays (see Section 8.2.2), as well as the "GraphBuild" stem of the graph building function name, are prefixed and suffixed by the prefptr and suffptr strings, respectively.

Return values

SCOTCH_graphDump returns 0 if the C source code corresponding to the graph structure has been successfully written to stream, and 1 else.

8.7.12 SCOTCH_graphExit

Synopsis

```
void SCOTCH_graphExit (SCOTCH_Graph *  grafptr)
scotchfgraphexit (doubleprecision (*)  grafdat)
```

Description

The SCOTCH_graphExit function frees the contents of a SCOTCH_Graph structure previously initialized by SCOTCH_graphInit. All subsequent calls to SCOTCH_graph routines other than SCOTCH_graphInit, using this structure as parameter, may yield unpredictable results.

8.7.13 SCOTCH_graphFree

Synopsis

```
void SCOTCH_graphFree (SCOTCH_Graph *  grafptr)
scotchfgraphfree (doubleprecision (*)  grafdat)
```

Description

The SCOTCH_graphFree function frees the graph data of a SCOTCH_Graph structure previously initialized by SCOTCH_graphInit, but preserves its internal data structures. This call is equivalent to a call to SCOTCH_graph Exit immediately followed by a call to SCOTCH_graphInit. Consequently, the given SCOTCH_Graph structure remains ready for subsequent calls to any routine of the LIBSCOTCH library.

8.7.14 SCOTCH_graphInduceList

Synopsis

```
int SCOTCH_graphInduceList (const SCOTCH_Graph *  orggrafptr,
                            SCOTCH_Num            vnumnbr,
                            SCOTCH_Num *          vnumtab,
                            SCOTCH_Graph *        indgrafptr)

scotchfgraphinducelist (doubleprecision (*)  orggrafdat,
                        integer*num          vnumnbr,
                        integernum (*)       vnumtab,
                        doubleprecision (*)  indgrafdat,
                        integer              ierr)
```

Description

The SCOTCH_graphInduceList routine computes an induced graph indgrafdat from the original graph orggrafdat. The vertices that are kept in the induced graph are the vnumnbr vertices whose based indices in the original graph are provided in the vnumtab array, in its first vnumnbr cells.

Return values

SCOTCH_graphInduceList returns 0 if the induced graph has been successfully computed, and 1 else.

8.7.15 SCOTCH_graphInducePart

Synopsis

```
int SCOTCH_graphInducePart (const SCOTCH_Graph *  orggrafptr,
                            SCOTCH_Num            vnumnbr,
                            SCOTCH_GraphPart2 *   parttab,
                            SCOTCH_GraphPart2     partval,
                            SCOTCH_Graph *        indgrafptr)

scotchfgraphinducepart (doubleprecision (*)  orggrafdat,
                        integer*num          vnumnbr,
                        characternum (*)     parttab,
                        characternum         partval,
                        doubleprecision (*)  indgrafdat,
                        integer              ierr)
```

Description

The SCOTCH_graphInducePart routine computes an induced graph indgrafdat from the original graph orggrafdat. The vertices that are kept in the induced graph are the vnumnbr vertices whose part number in the parttab array are equal to partval. The SCOTCH_GraphPart2 type, being a very small integer (most likely, an unsigned char), is assumed to hold only small values, e.g. 0 or 1.

Return values

SCOTCH_graphInducePart returns 0 if the induced graph has been successfully computed, and 1 else.

8.7.16 SCOTCH_graphInit

Synopsis

```
int SCOTCH_graphInit (SCOTCH_Graph *  grafptr)

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

Description

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

Return values

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

8.7.17 SCOTCH_graphLoad

Synopsis

```
int SCOTCH_graphLoad (SCOTCH_Graph *  grafptr,
                      FILE *          stream,
                      SCOTCH_Num      baseval,
                      SCOTCH_Num      flagval)

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

Description

The SCOTCH_graphLoad routine fills the SCOTCH_Graph structure pointed to by grafptr with the source graph description available from stream stream in the Scotch graph format (see section 6.1).

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.

Return values

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

8.7.18 SCOTCH_graphSave

Synopsis

```
int SCOTCH_graphSave (const SCOTCH_Graph *  grafptr,
                      FILE *                stream)

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

Description

The SCOTCH_graphSave routine saves the contents of the SCOTCH_Graph structure pointed to by grafptr to stream stream, in the Scotch graph format (see section 6.1).

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

8.7.19 SCOTCH_graphSize

Synopsis

```
void SCOTCH_graphSize (const SCOTCH_Graph *  grafptr,
                       SCOTCH_Num *          vertptr,
                       SCOTCH_Num *          edgeptr)

scotchfgraphsize (doubleprecision (*)  grafdat,
                  integer*num          vertnbr,
                  integer*num          edgenbr)
```

Description

The SCOTCH_graphSize routine fills the two areas of type SCOTCH_Num pointed to by vertptr and edgeptr with the number of vertices and arcs (that is, twice the number of edges) of the given graph pointed to by grafptr, respectively.

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_graphLoad routine, in order to allocate auxiliary arrays of proper sizes. If the whole structure of the graph is wanted, function SCOTCH_graphData should be preferred.

8.7.20 SCOTCH_graphSizeof

Synopsis

```
int SCOTCH_graphSizeof (void)
scotchfgraphsizeof (integer  size )
```

Description

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

8.7.21 SCOTCH_graphStat

Synopsis

```
void SCOTCH_graphStat (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)

scotchfgraphstat (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)
```

Description

The `SCOTCH_graphStat` routine produces some statistics regarding the 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.

8.8 High-level graph partitioning, mapping and clustering routines

The routines presented in this section provide high-level functionalities and free the user from the burden of calling in sequence several of the low-level routines described in the next section.

8.8.1 SCOTCH_graphMap

Synopsis

```
int SCOTCH_graphMap (const SCOTCH_Graph *  grafptr,
                     const SCOTCH_Arch *   archptr,
                     const SCOTCH_Strat *  straptr,
                     SCOTCH_Num *          parttab)

scotchfgraphmap (doubleprecision (*)  grafdat,
                 doubleprecision (*)  archdat,
                 doubleprecision (*)  stradat,
                 integer*num (*)      parttab,
                 integer              ierr)
```

Description

The SCOTCH_graphMap routine computes a mapping of the source graph structure pointed to by grafptr onto the target architecture pointed to by archptr, using the mapping strategy pointed to by straptr (as defined in Section 8.3.2), and returns the mapping data in the array pointed to by parttab.

The parttab array should have been previously allocated, of a size sufficient to hold as many SCOTCH_Num integers as there are vertices in the source graph.

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. This semantics aims at complying with standards such as MPI, in which process ranks start from 0.

When a variable-sized architecture is used (see Section 6.4.3) and a proper strategy is provided (see Section 8.16.2), the SCOTCH_graphMap routine can cluster the given graph by means of recursive bipartitioning. In this case, clusters are labeled according to a binary scheme: the part equal to the whole graph is numbered 1, its two bipartitioned descendants are labeled 2 and 3, the two descendants of part 2 are labeled 4 and 5, and so on. More generally, clusters are labeled such that the two descendants of any cluster i that has been split are labeled $2i$ and $2i + 1$.

Classical clustering strategies perform recursive bipartitioning of process graphs until some criterion is met: either parts become smaller than some size threshold, or edge density becomes higher than some ratio, etc. If graph mapping is performed using a variable-sized architecture and a classical mapping strategy, recursive bipartitioning will halt only when the load imbalance criterion allows for one of the bipartitioned parts to be empty (that is, most often, parts contains a single vertex).

Return values

SCOTCH_graphMap returns 0 if the mapping of the graph has been successfully computed, and 1 else. In this last case, the parttab array may however have been partially or completely filled, but its contents are not significant.

8.8.2 SCOTCH_graphMapFixed

Synopsis

```
int SCOTCH_graphMapFixed (const SCOTCH_Graph *  grafptr,
                          const SCOTCH_Arch *   archptr,
                          const SCOTCH_Strat *  straptr,
                          SCOTCH_Num *          parttab)

scotchfgraphmapfixed (doubleprecision (*)  grafdat,
                      doubleprecision (*)  archdat,
                      doubleprecision (*)  stradat,
                      integer*num (*)      parttab,
                      integer              ierr)
```

Description

The SCOTCH_graphMapFixed routine computes a mapping of the source graph structure pointed to by grafptr onto the target architecture pointed to by archptr, using the mapping strategy pointed to by straptr (as defined in Section 8.3.2), and fills the array pointed to by parttab with the mapping data regarding vertices which have not been pre-assigned by the user.

The parttab array should have been previously allocated, of a size sufficient to hold as many SCOTCH_Num integers as there are vertices in the source graph. It must also have been filled in advance by the user, with data indicating whether vertices have been already pre-assigned to a fixed position or are to be processed by the routine. In each cell of the parttab array, a value of -1 indicates that the vertex is movable, while a value between 0 and the number of target vertices minus 1 indicates that the vertex has been pre-assigned to the given part.

On return, every cell of the mapping array that contained a -1 will hold 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. This semantics aims at complying with standards such as MPI, in which process ranks start from 0.

Return values

SCOTCH_graphMapFixed returns 0 if the mapping of the graph has been successfully computed, and 1 else. In this last case, the parttab array may however have been partially or completely filled, but its contents are not significant.

8.8.3 SCOTCH_graphPart

Synopsis

```
int SCOTCH_graphPart (const SCOTCH_Graph *  grafptr,
                      const SCOTCH_Num      partnbr,
                      const SCOTCH_Strat *  straptr,
                      SCOTCH_Num *          parttab)

scotchfgraphpart (doubleprecision (*)  grafdat,
                  integer*num          partnbr,
                  doubleprecision (*)  stradat,
                  integer*num (*)      parttab,
                  integer              ierr)
```

Description

The SCOTCH_graphPart routine computes an edge-separated partition, into partnbr parts, of the source graph structure pointed to by grafptr, using the graph edge partitioning strategy pointed to by stratptr (as defined in Section 8.3.2), and returns the partition data in the array pointed to by parttab.

The parttab array should have been previously allocated, of a size sufficient to hold as many SCOTCH_Num integers as there are vertices in the source graph.

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 **partnbr**− 1. This semantics aims at complying with standards such as MPI, in which process ranks start from 0.

Return values

SCOTCH_graphPart returns 0 if the graph partition has been successfully computed, and 1 else. In the latter case, the parttab array may however have been partially or completely filled, but its contents are not significant.

8.8.4 SCOTCH_graphPartFixed

Synopsis

```
int SCOTCH_graphPartFixed (const SCOTCH_Graph *  grafptr,
                           const SCOTCH_Num      partnbr,
                           const SCOTCH_Strat *  straptr,
                           SCOTCH_Num *          parttab)

scotchfgraphpartfixed (doubleprecision (*)  grafdat,
                       integer*num          partnbr,
                       doubleprecision (*)  stradat,
                       integer*num (*)      parttab,
                       integer              ierr)
```

Description

The SCOTCH_graphPartFixed routine computes an edge-separated partition, into partnbr parts, of the source graph structure pointed to by

grafptr, using the graph edge partitioning strategy pointed to by stratptr (as defined in Section 8.3.2), and fills the array pointed to by parttab with the partitioning data regarding vertices which have not been pre-assigned by the user.

The parttab array should have been previously allocated, of a size sufficient to hold as many SCOTCH_Num integers as there are vertices in the source graph. It must also have been filled in advance by the user, with data indicating whether vertices have been already pre-assigned to a fixed position or are to be processed by the routine. In each cell of the parttab array, a value of -1 indicates that the vertex is movable, while a value between 0 and the number of target vertices minus 1 indicates that the vertex has been pre-assigned to the given part.

On return, every cell of the mapping array that contained a -1 will hold the number of the target vertex to which the corresponding source vertex is assigned. The numbering of target values is *not* based: target vertices are numbered from 0 to the number of target vertices minus 1. This semantics aims at complying with standards such as MPI, in which process ranks start from 0.

Return values

SCOTCH_graphPartFixed returns 0 if the graph partition has been successfully computed, and 1 else. In the latter case, the parttab array may however have been partially or completely filled, but its contents are not significant.

8.8.5 SCOTCH_graphPartOvl

Synopsis

```
int SCOTCH_graphPartOvl (const SCOTCH_Graph *  grafptr,
                         const SCOTCH_Num      partnbr,
                         const SCOTCH_Strat *  straptr,
                         SCOTCH_Num *          parttab)

scotchfgraphpartovl (doubleprecision (*)  grafdat,
                     integer*num          partnbr,
                     doubleprecision (*)  stradat,
                     integer*num (*)      parttab,
                     integer              ierr)
```

Description

The SCOTCH_graphPartOvl routine computes an overlapped vertex-separated partition, into partnbr parts, of the source graph structure pointed to by grafptr, using the graph vertex partitioning with overlap strategy pointed to by stratptr (as defined in Section 8.3.4), and returns the partition data in the array pointed to by parttab.

The parttab array should have been previously allocated, of a size sufficient to hold as many SCOTCH_Num integers as there are vertices in the source graph.

On return, every array cell holds the number of the part to which the corresponding vertex is mapped. Regular parts are numbered from 0 to `partnbr`−1, and separator vertices are labeled with part number `-1`.

While `SCOTCH_graphMap` and `SCOTCH_graphPart` are based on edge partitioning methods, `SCOTCH_graphPartOvl` relies on a completely distinct set of routines to compute vertex separators. This is why `SCOTCH_graphPartOvl` requires strategy strings of a different kind, created by the `SCOTCH_stratGraphPartOvl*` routines only (see Sections 8.16.5 and 8.16.6).

Return values

`SCOTCH_graphPartOvl` returns 0 if the partition of the graph has been successfully computed, and 1 else. In the latter case, the `parttab` array may however have been partially or completely filled, but its contents are not significant.

8.8.6 `SCOTCH_graphPartOvlView`

Synopsis

```
int SCOTCH_graphPartOvlView (const SCOTCH_Graph *  grafptr,
                             SCOTCH_Num            partnbr,
                             SCOTCH_Num *          parttab,
                             FILE *                stream)

scotchfgraphpartovlview (doubleprecision (*)  grafdat,
                         integer*num          partnbr,
                         integer*num (*)      parttab,
                         integer              fildes,
                         integer              ierr)
```

Description

The `SCOTCH_graphPartOvlView` routine writes to `stream` the previously computed overlapped vertex-separated partition of the source graph `grafptr` into `partnbr` parts, as passed in `parttab`. See also `SCOTCH_graphPartOvl`, Section .

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

8.8.7 `SCOTCH_graphRemap`

Synopsis

```
int SCOTCH_graphRemap (const SCOTCH_Graph *  grafptr,
                       const SCOTCH_Arch *   archptr,
                       const SCOTCH_Num *    parotab,
                       const double          emraval,
                       const SCOTCH_Num *    vmlotab,
                       const SCOTCH_Strat *  straptr,
                       SCOTCH_Num *          parttab)

scotchfgraphremap (doubleprecision (*)  grafdat,
                   doubleprecision (*)  archdat,
                   integer*num (*)      parotab,
                   doubleprecision      emraval,
                   integer*num (*)      vmlotab,
                   doubleprecision (*)  stradat,
                   integer*num (*)      parttab,
                   integer              ierr)
```

Description

The SCOTCH_graphRemap routine computes a remapping of the source graph structure pointed to by grafptr onto the target architecture pointed to by archptr, based on the old partition array pointed to by parotab, using the mapping strategy pointed to by straptr (as defined in Section 8.3.2), and returns the mapping data in the array pointed to by parttab.

The parotab array stores the old partition that is used to compute migration costs. Every cell contains values from 0 to the number of target vertices minus 1, or -1 for vertices that did not belong to the old partition (e.g., vertices newly created by graph adaptation, which can be placed at no cost before their associated data is interpolated).

With every source graph vertex is associated an individual integer migration cost, stored in the vmlotab array. These costs are accounted for in the communication cost function to minimize as multiples of the individual migration cost emraval. Since this value is provided as a floating point number, migration costs can be set as fractions or as non-integer multiples of the cut metric communication costs stored as integer edge loads.

The parttab array should have been previously allocated, of a size sufficient to hold as many SCOTCH_Num integers as there are vertices in the source graph.

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. This semantics aims at complying with standards such as MPI, in which process ranks start from 0.

Return values

SCOTCH_graphRemap returns 0 if the mapping of the graph has been successfully computed, and 1 else. In this last case, the parttab array may however have been partially or completely filled, but its contents are not significant.

8.8.8 SCOTCH_graphRemapFixed

Synopsis

```
int SCOTCH_graphRemapFixed (const SCOTCH_Graph *  grafptr,
                            const SCOTCH_Arch *   archptr,
                            const SCOTCH_Num *    parotab,
                            const double          emraval,
                            const SCOTCH_Num *    vmlotab,
                            const SCOTCH_Strat *  straptr,
                            SCOTCH_Num *          parttab)

scotchfgraphremapfixed (doubleprecision (*)  grafdat,
                        doubleprecision (*)  archdat,
                        integer*num (*)      parotab,
                        doubleprecision      emraval,
                        integer*num (*)      vmlotab,
                        doubleprecision (*)  stradat,
                        integer*num (*)      parttab,
                        integer              ierr)
```

Description

The SCOTCH_graphRemapFixed routine computes a remapping of the source graph structure pointed to by grafptr onto the target architecture pointed to by archptr, based on the old partition array pointed to by parotab, using the mapping strategy pointed to by straptr (as defined in Section 8.3.2), and fills the array pointed to by parttab with the mapping data regarding vertices which have not been pre-assigned by the user.

The parotab array stores the old partition that is used to compute migration costs. Every cell contains values from 0 to the number of target vertices minus 1, or -1 for vertices that did not belong to the old partition (e.g., vertices newly created by graph adaptation, which can be placed at no cost before their associated data is interpolated).

With every source graph vertex is associated an individual integer migration cost, stored in the vmlotab array. These costs are accounted for in the communication cost function to minimize as multiples of the individual migration cost emraval. Since this value is provided as a floating point number, migration costs can be set as fractions or as non-integer multiples of the cut metric communication costs stored as integer edge loads.

The parttab array should have been previously allocated, of a size sufficient to hold as many SCOTCH_Num integers as there are vertices in the source graph. It must also have been filled in advance by the user, with data indicating whether vertices have been already pre-assigned to a fixed position or are to be processed by the routine. In each cell of the parttab array, a value of -1 indicates that the vertex is movable, while a value between 0 and the number of target vertices minus 1 indicates that the vertex has been pre-assigned to the given part.

On return, every cell of the mapping array that contained a -1 will hold 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. This semantics aims at complying with standards such as MPI, in which process ranks start from 0.

Return values

`SCOTCH_graphRemapFixed` returns 0 if the mapping of the graph has been successfully computed, and 1 else. In this last case, the `parttab` array may however have been partially or completely filled, with some −1's removed, but its contents are not significant.

8.8.9 `SCOTCH_graphRepart`

Synopsis

```
int SCOTCH_graphRepart (const SCOTCH_Graph *  grafptr,
                        const SCOTCH_Num      partnbr,
                        const SCOTCH_Num *    parotab,
                        const double          emraval,
                        const SCOTCH_Num *    vmlotab,
                        const SCOTCH_Strat *  straptr,
                        SCOTCH_Num *          parttab)

scotchfgraphrepart (doubleprecision (*)  grafdat,
                    integer*num          partnbr,
                    integer*num (*)      parotab,
                    doubleprecision      emraval,
                    integer*num (*)      vmlotab,
                    doubleprecision (*)  stradat,
                    integer*num (*)      parttab,
                    integer              ierr)
```

Description

The `SCOTCH_graphRepart` routine computes an edge-separated repartition, into `partnbr` parts, of the source graph structure pointed to by `grafptr`, based on the old partition array pointed to by `parotab`, using the partitioning strategy pointed to by `straptr` (as defined in Section 8.3.2), and returns the partition data in the array pointed to by `parttab`.

The `parotab` array stores the old partition that is used to compute migration costs. Every cell contains values from 0 to the number of target vertices minus 1, or −1 for vertices that did not belong to the old partition (e.g., vertices newly created by graph adaptation, which can be assigned to any part at no cost before their associated data is interpolated).

With every source graph vertex is associated an individual integer migration cost, stored in the `vmlotab` array. These costs are accounted for in the communication cost function to minimize as multiples of the individual migration cost `emraval`. Since this value is provided as a floating point number, migration costs can be set as fractions or as non-integer multiples of the cut metric communication costs stored as integer edge loads.

The parttab array should have been previously allocated, of a size sufficient to hold as many SCOTCH_Num integers as there are vertices in the source graph.

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. This semantics aims at complying with standards such as MPI, in which process ranks start from 0.

Return values

SCOTCH_graphRepart returns 0 if the graph partition has been successfully computed, and 1 else. In the latter case, the parttab array may however have been partially or completely filled, but its contents are not significant.

8.8.10 SCOTCH_graphRepartFixed

Synopsis

```
int SCOTCH_graphRepartFixed (const SCOTCH_Graph *  grafptr,
                             const SCOTCH_Num      partnbr,
                             const SCOTCH_Num *    parotab,
                             const double          emraval,
                             const SCOTCH_Num *    vmlotab,
                             const SCOTCH_Strat *  straptr,
                             SCOTCH_Num *          parttab)

scotchfgraphrepartfixed (doubleprecision (*)  grafdat,
                         integer*num          partnbr,
                         integer*num (*)      parotab,
                         doubleprecision      emraval,
                         integer*num (*)      vmlotab,
                         doubleprecision (*)  stradat,
                         integer*num (*)      parttab,
                         integer              ierr)
```

Description

The SCOTCH_graphRepartFixed routine computes an edge-separated repartition, into partnbr parts, of the source graph structure pointed to by grafptr, based on the old partition array pointed to by parotab, using the partitioning strategy pointed to by straptr (as defined in Section 8.3.2), and fills the array pointed to by parttab with the mapping data regarding vertices which have not been pre-assigned by the user.

The parotab array stores the old partition that is used to compute migration costs. Every cell contains values from 0 to the number of target vertices minus 1, or -1 for vertices that did not belong to the old partition (e.g., vertices newly created by graph adaptation, which can be assigned to any part at no cost before their associated data is interpolated).

With every source graph vertex is associated an individual integer migration cost, stored in the vmlotab array. These costs are accounted for in the com-

munication cost function to minimize as multiples of the individual migration cost `emraval`. Since this value is provided as a floating point number, migration costs can be set as fractions or as non-integer multiples of the cut metric communication costs stored as integer edge loads.

The `parttab` array should have been previously allocated, of a size sufficient to hold as many `SCOTCH_Num` integers as there are vertices in the source graph. It must also have been filled in advance by the user, with data indicating whether vertices have been already pre-assigned to a fixed position or are to be processed by the routine. In each cell of the `parttab` array, a value of -1 indicates that the vertex is movable, while a value between 0 and the number of target vertices minus 1 indicates that the vertex has been pre-assigned to the given part.

On return, every cell of the mapping array that contained a -1 will hold 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. This semantics aims at complying with standards such as MPI, in which process ranks start from 0.

Return values

`SCOTCH_graphRepartFixed` returns 0 if the graph partition has has been successfully computed, and 1 else. In this last case, the `parttab` array may however have been partially or completely filled, with some -1's removed, but its contents are not significant.

8.9 Low-level graph partitioning, mapping and clustering routines

All of the following routines operate on a `SCOTCH_Mapping` structure that contains references to the partition and mapping arrays to be filled during the mapping or remapping process.

8.9.1 `SCOTCH_graphMapCompute`

Synopsis

```
int SCOTCH_graphMapCompute (const SCOTCH_Graph *  grafptr,
                            SCOTCH_Mapping *      mappptr,
                            const SCOTCH_Strat *  straptr)

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

Description

The `SCOTCH_graphMapCompute` routine computes a mapping on the given `SCOTCH_Mapping` structure pointed to by `mappptr` using the mapping strategy pointed to by `stratptr`.

On return, every cell of the mapping array defined by SCOTCH_mapInit 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.

Return values

SCOTCH_graphMapCompute returns 0 if the mapping has been successfully computed, and 1 else. In this latter case, the mapping array may however have been partially or completely filled, but its contents are not significant.

8.9.2 SCOTCH_graphMapExit

Synopsis

```
void SCOTCH_graphMapExit (const SCOTCH_Graph *  grafptr,
                          SCOTCH_Mapping *      mappptr)

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

Description

The SCOTCH_graphMapExit function frees the contents of a SCOTCH_Mapping structure previously initialized by SCOTCH_graphMapInit. All subsequent calls to SCOTCH_graphMap* routines other than SCOTCH_graphMapInit, using this structure as parameter, may yield unpredictable results.

8.9.3 SCOTCH_graphMapFixedCompute

Synopsis

```
int SCOTCH_graphMapFixedCompute (const SCOTCH_Graph *  grafptr,
                                 SCOTCH_Mapping *      mappptr,
                                 const SCOTCH_Strat *  straptr)

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

Description

The SCOTCH_graphMapFixedCompute routine computes a mapping on the given SCOTCH_Mapping structure pointed to by mappptr using the mapping strategy pointed to by stratptr. The mapping must have been built so that its partition array has been filled in advance by the user, with data indicating whether vertices have been already pre-assigned to a fixed position or are to be processed by the routine. In each cell of the parttab array, a value of -1 indicates that the vertex is movable, while a value between 0 and the number

of target vertices minus 1 indicates that the vertex has been pre-assigned to the given part.

On return, every cell of the mapping array defined by SCOTCH_mapInit that contained a -1 will hold 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.

Return values

SCOTCH_graphMapFixedCompute returns 0 if the mapping has been successfully computed, and 1 else. In this latter case, the mapping array may however have been partially or completely filled, with some -1's removed, but its contents are not significant.

8.9.4 SCOTCH_graphMapInit

Synopsis

```
int SCOTCH_graphMapInit (const SCOTCH_Graph *  grafptr,
                         SCOTCH_Mapping *      mappptr,
                         const SCOTCH_Arch *   archptr,
                         SCOTCH_Num *          parttab)

scotchfgraphmapinit (doubleprecision (*)  grafdat,
                     doubleprecision (*)  mappdat,
                     doubleprecision (*)  archdat,
                     integer*num (*)      parttab,
                     integer              ierr)
```

Description

The SCOTCH_graphMapInit routine fills the 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_graphMapCompute, using this mapping structure as parameter, will place mapping results in field parttab.

parttab is the pointer to an array of as many SCOTCH_Nums as there are vertices in the 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_Mapping structure. When the mapping structure is no longer of use, call function SCOTCH_graph MapExit to free its internal structures.

Return values

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

8.9.5 SCOTCH_graphMapLoad

Synopsis

```
int SCOTCH_graphMapLoad (const SCOTCH_Graph *  grafptr,
                         SCOTCH_Mapping *      mappptr,
                         FILE *                stream)

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

Description

The SCOTCH_graphMapLoad routine fills the SCOTCH_Mapping structure pointed to by mappptr with the mapping data available in the Scotch mapping format (see section 6.5) from stream stream. If the source graph has vertex labels attached to its vertices, mapping indices in the input stream are assumed to be vertex labels as well.

Users willing to have subsequent access to the partition data rather than to fill an opaque SCOTCH_Mapping structure are invited to use the SCOTCH_graphTabLoad routine instead.

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_graphMapLoad returns 0 if the mapping structure has been successfully loaded from stream, and 1 else.

8.9.6 SCOTCH_graphMapSave

Synopsis

```
int SCOTCH_graphMapSave (const SCOTCH_Graph *    grafptr,
                         const SCOTCH_Mapping *  mappptr,
                         FILE *                  stream)

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

Description

The SCOTCH_graphMapSave routine saves the contents of the SCOTCH_Mapping structure pointed to by mappptr to stream stream, in the Scotch mapping format (see section 6.5).

Users willing to save a partition data array rather than an opaque SCOTCH_Mapping structure are invited to use the SCOTCH_graphTabSave routine instead.

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

8.9.7 SCOTCH_graphMapView

Synopsis

```
int SCOTCH_graphMapView (const SCOTCH_Graph *    grafptr,
                         const SCOTCH_Mapping *  mappptr,
                         FILE *                  stream)

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

Description

The SCOTCH_graphMapView 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_graphMapView returns 0 if the data has been successfully written to stream, and 1 else.

8.9.8 SCOTCH_graphRemapCompute

Synopsis

```
int SCOTCH_graphRemapCompute (const SCOTCH_Graph *  grafptr,
                              SCOTCH_Mapping *      mappptr,
                              SCOTCH_Mapping *      mapoptr,
                              const double          emraval,
                              const SCOTCH_Num *    vmlotab,
                              const SCOTCH_Strat *  straptr)
```

```
scotchfgraphremapcompute (doubleprecision (*)  grafdat,
                          doubleprecision (*)  mappdat,
                          doubleprecision (*)  mapodat,
                          doubleprecision      emraval,
                          integer*num (*)      vmlotab,
                          doubleprecision (*)  stradat,
                          integer              ierr)
```

Description

The SCOTCH_graphRemapCompute routine computes a mapping on the given SCOTCH_Mapping structure pointed to by mappptr, using the mapping strategy pointed to by stratptr, and accounting for migration costs computed based on the already computed partition pointed to by mapoptr. This partition should have been created from the same graph and target architecture as the one pointer to by mappptr.

With every source graph vertex is associated an individual integer migration cost, stored in the vmlotab array. These costs are accounted for in the communication cost function to minimize as multiples of the individual migration cost emraval. Since this value is provided as a floating point number, migration costs can be set as fractions or as non-integer multiples of the cut metric communication costs stored as integer edge loads.

On return, every cell of the new mapping array defined by SCOTCH_mapInit 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.

Return values

SCOTCH_graphRemapCompute returns 0 if the remapping has been successfully computed, and 1 else. In this latter case, the mapping array may however have been partially or completely filled, but its contents are not significant.

8.9.9 SCOTCH_graphRemapFixedCompute

Synopsis

```
int SCOTCH_graphRemapFixedCompute (const SCOTCH_Graph *  grafptr,
                                   SCOTCH_Mapping *      mappptr,
                                   SCOTCH_Mapping *      mapoptr,
                                   const double          emraval,
                                   const SCOTCH_Num *    vmlotab,
                                   const SCOTCH_Strat *  straptr)

scotchfgraphremapfixedcompute (doubleprecision (*)  grafdat,
                               doubleprecision (*)  mappdat,
                               doubleprecision (*)  mapodat,
                               doubleprecision      emraval,
                               integer*num (*)      vmlotab,
                               doubleprecision (*)  stradat,
                               integer              ierr)
```

Description

The SCOTCH_graphRemapFixedCompute routine computes a mapping on the given SCOTCH_Mapping structure pointed to by mappptr, using the mapping strategy pointed to by stratptr, and accounting for migration costs computed based on the already computed partition pointed to by mapoptr. This partition should have been created from the same graph and target architecture as the one pointer to by mappptr.

The partition array of the mapping pointed to by mappptr must have been filled in advance by the user, with data indicating whether vertices have been already pre-assigned to a fixed position or are to be processed by the routine. A value of -1 indicates that the vertex is movable, while a value between 0 and the number of target vertices minus 1 indicates that the vertex has been pre-assigned to the given part.

With every source graph vertex is associated an individual integer migration cost, stored in the vmlotab array. These costs are accounted for in the communication cost function to minimize as multiples of the individual migration cost emraval. Since this value is provided as a floating point number, migration costs can be set as fractions or as non-integer multiples of the cut metric communication costs stored as integer edge loads.

On return, every cell of the new mapping array defined by SCOTCH_mapInit that contained a -1 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.

Return values

SCOTCH_graphRemapFixedCompute returns 0 if the remapping has been successfully computed, and 1 else. In this latter case, the mapping array may however have been partially or completely filled, with some -1's removed, but its contents are not significant.

8.9.10 SCOTCH_graphTabLoad

Synopsis

```
int SCOTCH_graphTabLoad (const SCOTCH_Graph *  grafptr,
                         SCOTCH_Num *          parttab,
                         FILE *                stream)

scotchfgraphtabload (doubleprecision (*)  grafdat,
                     integer*num (*)      parttab,
                     integer              fildes,
                     integer              ierr)
```

Description

The SCOTCH_graphTabLoad routine fills the part array pointed to by parttab with the mapping data available in the Scotch mapping format (see section 6.5) from stream stream.

This routine allows users to fill plain partition arrays rather than opaque mapping structures, as routine SCOTCH_graphMapLoad does.

The parttab array should have been previously allocated, of a size sufficient to hold as many SCOTCH_Num integers as there are vertices in the source graph. Upon completion, array cells contain the indices of the parts to which vertices belong according to the input mapping stream, or −1 if they were not mentioned in the stream. If the source graph has vertex labels attached to its vertices, mapping indices in the input stream are assumed to be vertex labels as well.

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_graphTabLoad returns 0 if the part array has been successfully loaded from stream, and 1 else.

8.9.11 SCOTCH_graphTabSave

Synopsis

```
int SCOTCH_graphTabSave (const SCOTCH_Graph *  grafptr,
                         SCOTCH_Num *          parttab,
                         FILE *                stream)

scotchfgraphtabsave (doubleprecision (*)  grafdat,
                     integer*num (*)      parttab,
                     integer              fildes,
                     integer              ierr)
```

Description

The SCOTCH_graphTabSave routine saves to stream stream the contents of the part array pointed to by parttab, on the form of mapping data in the Scotch mapping format (see section 6.5).

This routine allows users to save plain partition arrays rather than opaque mapping structures, as routine SCOTCH_graphMapSave does.

Upon completion, the produced mapping file contain the indices of the parts to which vertices belong according to the given part array. If the source graph has vertex labels attached to its vertices, mapping indices in the output stream are replaced by the vertex labels.

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_graphTabSave returns 0 if the part array has been successfully saved to stream, and 1 else.

8.10 High-level graph ordering routines

This routine provides high-level functionality and frees the user from the burden of calling in sequence several of the low-level routines described in the next section.

8.10.1 SCOTCH_graphOrder

Synopsis

```
int SCOTCH_graphOrder (const SCOTCH_Graph *  grafptr,
                       const SCOTCH_Strat *  straptr,
                       SCOTCH_Num *          permtab,
                       SCOTCH_Num *          peritab,
                       SCOTCH_Num *          cblkptr,
                       SCOTCH_Num *          rangtab,
                       SCOTCH_Num *          treetab)

scotchfgraphorder (doubleprecision (*)  grafdat,
                   doubleprecision (*)  stradat,
                   integer*num (*)      permtab,
                   integer*num (*)      peritab,
                   integer*num          cblknbr,
                   integer*num (*)      rangtab,
                   integer*num (*)      treetab,
                   integer              ierr)
```

Description

The SCOTCH_graphOrder routine computes a block ordering of the unknowns of the symmetric sparse matrix the adjacency structure of which is represented by the source graph structure pointed to by grafptr, using the ordering strategy pointed to by stratptr, and returns ordering data in the scalar pointed to by cblkptr and the four arrays permtab, peritab, rangtab and treetab.

The permtab, peritab, rangtab and treetab arrays should have been previously allocated, of a size sufficient to hold as many SCOTCH_Num integers as there are vertices in the source graph, plus one in the case of rangtab. Any of the 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 in these fields will have the same effect.

On return, permtab holds the direct permutation of the unknowns, that is, vertex i of the original graph has index permtab[i] in the reordered graph, while peritab holds the inverse permutation, that is, vertex i in the reordered graph had index peritab[i] in the original graph. All of these indices are numbered according to the base value of the source graph: permutation indices are numbered from baseval to **vertnbr** + **baseval** − 1,

that is, from 0 to `vertnbr` − 1 if the graph base is 0, and from 1 to `vertnbr` if the graph base is 1.

The three other result fields, `*cblkptr`, `rangtab` and `treetab`, contain data related to the block structure. `*cblkptr` holds the number of column blocks of the produced ordering, and `rangtab` holds the starting indices of each of the permuted column blocks, in increasing order, so that column block i starts at index `rangtab[`i`]` and ends at index (`rangtab[`i+1]−1), inclusive, in the new ordering. `treetab` holds the separators tree structure, that is, `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. Please refer to Section 8.2.5 for more information.

Return values

`SCOTCH_graphOrder` returns 0 if the ordering of the graph has been successfully computed, and 1 else. In this last case, the `rangtab`, `permtab`, and `peritab` arrays may however have been partially or completely filled, but their contents are not significant.

8.11 Low-level graph ordering routines

All of the following routines operate on a `SCOTCH_Ordering` structure that contains references to the permutation arrays to be filled during the graph ordering process.

8.11.1 `SCOTCH_graphOrderCheck`

Synopsis

```
int SCOTCH_graphOrderCheck (const SCOTCH_Graph *     grafptr,
                            const SCOTCH_Ordering *  ordeptr)

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

Description

The `SCOTCH_graphOrderCheck` routine checks the consistency of the given `SCOTCH_Ordering` structure pointed to by `ordeptr`.

Return values

`SCOTCH_graphOrderCheck` returns 0 if ordering data are consistent, and 1 else.

8.11.2 `SCOTCH_graphOrderCompute`

Synopsis

```
int SCOTCH_graphOrderCompute (const SCOTCH_Graph *  grafptr,
                              SCOTCH_Ordering *     ordeptr,
                              const SCOTCH_Strat *  straptr)
```

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

Description

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

On return, the ordering structure holds a block ordering of the given graph (see section 8.11.5 for a description of the ordering fields).

Return values

SCOTCH_graphOrderCompute 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.

8.11.3 SCOTCH_graphOrderComputeList

Synopsis

```
int SCOTCH_graphOrderComputeList (const SCOTCH_Graph *  grafptr,
                                  SCOTCH_Ordering *     ordeptr,
                                  SCOTCH_Num            listnbr,
                                  SCOTCH_Num *          listtab,
                                  const SCOTCH_Strat *  straptr)

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

Description

The SCOTCH_graphOrderComputeList routine computes a block ordering of a subgraph of the graph structure pointed to by grafptr, using the ordering strategy pointed to by stratptr, and stores its result in the ordering structure pointed to by ordeptr. The induced subgraph is described by means of a vertex list: listnbr holds the number of vertices to keep in the induced subgraph, the indices of which are given, in any order, in the listtab array.

On return, the ordering structure holds a block ordering of the induced subgraph (see section 8.2.5 for a description of the ordering fields). To compute this ordering, graph ordering methods such as the minimum degree and minimum fill methods will base on the original degree of the induced graph vertices,

their non-induced neighbors being considered as halo vertices (see Section 4.4 for more information on halo vertices).

Because an ordering always refers to the full graph, the ordering computed by SCOTCH_graphOrderComputeList is divided into two distinct parts: the induced graph vertices are ordered by applying to the induced graph the strategy provided by the stratptr parameter, while non-induced vertex 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. The separation tree yielded by SCOTCH_graphOrderComputeList reflects this property: it is made of two branches, the first one corresponding to the induced subgraph, and the second one to the remaining vertices. Since these two subgraphs are not considered to be connected, both will have their own root, represented by a −1 value in the treetab array of the ordering.

Return values

SCOTCH_graphOrderComputeList 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.

8.11.4 SCOTCH_graphOrderExit

Synopsis

```
void SCOTCH_graphOrderExit (const SCOTCH_Graph *  grafptr,
                            SCOTCH_Ordering *     ordeptr)
scotchfgraphorderexit (doubleprecision (*)  grafdat,
                       doubleprecision (*)  ordedat)
```

Description

The SCOTCH_graphOrderExit function frees the contents of a SCOTCH_Ordering structure previously initialized by SCOTCH_graphOrderInit. All subsequent calls to SCOTCH_graphOrder* routines other than SCOTCH_graphOrderInit, using this structure as parameter, may yield unpredictable results.

8.11.5 SCOTCH_graphOrderInit

Synopsis

```
int SCOTCH_graphOrderInit (const SCOTCH_Graph *  grafptr,
                           SCOTCH_Ordering *     ordeptr,
                           SCOTCH_Num *          permtab,
                           SCOTCH_Num *          peritab,
                           SCOTCH_Num *          cblkptr,
                           SCOTCH_Num *          rangtab,
                           SCOTCH_Num *          treetab)
```

```
scotchfgraphorderinit (doubleprecision (*)  grafdat,
                       doubleprecision (*)  ordedat,
                       integer*num (*)      permtab,
                       integer*num (*)      peritab,
                       integer*num          cblknbr,
                       integer*num (*)      rangtab,
                       integer*num (*)      treetab,
                       integer              ierr)
```

Description

The SCOTCH_graphOrderInit routine fills the ordering structure pointed to by ordeptr with all of the data that are passed to it. Thus, all subsequent calls to ordering routines such as SCOTCH_graphOrderCompute, using this ordering structure as parameter, will place ordering results in fields permtab, peritab, *cblkptr, rangtab or treetab, if they are not set to NULL.

permtab is the ordering permutation array, of size **vertnbr**, peritab is the inverse ordering permutation array, of size **vertnbr**, 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 **vertnbr**+1, and treetab is the array holding the structure of the separators tree, of size **vertnbr**. See the above manual page of SCOTCH_graphOrder, as well as section 8.2.5, for an explanation of the semantics of all of these fields.

The SCOTCH_graphOrderInit routine should be the first function to be called upon a SCOTCH_Ordering structure for ordering graphs. When the ordering structure is no longer of use, the SCOTCH_graphOrderExit function must be called, in order to to free its internal structures.

Return values

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

8.11.6 SCOTCH_graphOrderLoad

Synopsis

```
int SCOTCH_graphOrderLoad (const SCOTCH_Graph *  grafptr,
                           SCOTCH_Ordering *     ordeptr,
                           FILE *                stream)

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

Description

The SCOTCH_graphOrderLoad routine fills the SCOTCH_Ordering structure pointed to by ordeptr with the ordering data available in the Scotch ordering format (see section 6.6) from 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 ordering file.

Return values

SCOTCH_graphOrderLoad returns 0 if the ordering structure has been successfully loaded from stream, and 1 else.

8.11.7 SCOTCH_graphOrderSave

Synopsis

```
int SCOTCH_graphOrderSave (const SCOTCH_Graph *     grafptr,
                           const SCOTCH_Ordering *  ordeptr,
                           FILE *                   stream)

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

Description

The SCOTCH_graphOrderSave routine saves the contents of the SCOTCH_Ordering structure pointed to by ordeptr to stream stream, in the Scotch ordering format (see section 6.6).

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.

Return values

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

8.11.8 SCOTCH_graphOrderSaveMap

Synopsis

```
int SCOTCH_graphOrderSaveMap (const SCOTCH_Graph *     grafptr,
                              const SCOTCH_Ordering *  ordeptr,
                              FILE *                   stream)

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

Description

The SCOTCH_graphOrderSaveMap routine saves the block partitioning data associated with the SCOTCH_Ordering structure pointed to by ordeptr to stream stream, in the Scotch mapping format (see section 6.5). 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 (see Section 7.4.12) to produce pictures showing the different separators and blocks.

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

8.11.9 SCOTCH_graphOrderSaveTree

Synopsis

```
int SCOTCH_graphOrderSaveTree (const SCOTCH_Graph *     grafptr,
                               const SCOTCH_Ordering *  ordeptr,
                               FILE *                   stream)

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

Description

The SCOTCH_graphOrderSaveTree routine saves the tree hierarchy information associated with the SCOTCH_Ordering structure pointed to by ordeptr to stream stream.

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

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 tree mapping file.

Return values

SCOTCH_graphOrderSaveTree returns 0 if the separators tree structure has been successfully written to stream, and 1 else.

8.12 Mesh handling routines

8.12.1 SCOTCH_meshAlloc

Synopsis

```
SCOTCH_Mesh * SCOTCH_meshAlloc (void)
```

Description

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

Return values

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

8.12.2 SCOTCH_meshBuild

Synopsis

```
int SCOTCH_meshBuild (SCOTCH_Mesh *       meshptr,
                      const SCOTCH_Num    velmbas,
                      const SCOTCH_Num    vnodbas,
                      const SCOTCH_Num    velmnbr,
                      const SCOTCH_Num    vnodnbr,
                      const SCOTCH_Num *  verttab,
                      const SCOTCH_Num *  vendtab,
                      const SCOTCH_Num *  velotab,
                      const SCOTCH_Num *  vnlotab,
                      const SCOTCH_Num *  vlbltab,
                      const SCOTCH_Num    edgenbr,
                      const SCOTCH_Num *  edgetab)
scotchfmeshbuild (doubleprecision (*)  meshdat,
                  integer*num          velmbas,
                  integer*num          vnodbas,
                  integer*num          velmnbr,
                  integer*num          vnodnbr,
                  integer*num (*)      verttab,
                  integer*num (*)      vendtab,
                  integer*num (*)      velotab,
                  integer*num (*)      vnlotab,
                  integer*num (*)      vlbltab,
                  integer*num          edgenbr,
                  integer*num (*)      edgetab,
                  integer*num          ierr)
```

Description

The SCOTCH_meshBuild routine fills the source mesh structure pointed to by meshptr with all of the data that is passed to it.

velmbas and vnodbas are the base values for the element and node vertices, respectively. velmnbr and vnodnbr are the number of element and node vertices, respectively, such that either **velmbas** + **velmnbr** = **vnodnbr** or **vnodbas** + **vnodnbr** = **velmnbr** holds, and typically min(**velmbas**, **vnodbas**) is 0 for structures built from C and 1 for structures built from Fortran. verttab is the adjacency index array, of size (**velmnbr** + **vnodnbr** + 1) if the edge array is compact (that is, if vendtab equals (verttab + 1) or NULL), or of size (**velmnbr** + **vnodnbr**) else. vendtab is the adjacency end index array, of size (**velmnbr** + **vnodnbr**) if it is disjoint from verttab. velotab is the element vertex load array, of size velmnbr if it exists. vnlotab is the node vertex load array, of size vnodnbr if it exists. vlbltab is the vertex label array, of size (**velmnbr** + **vnodnbr**) if it exists. edgenbr is the number of arcs (that is, twice the number of edges). edgetab is the adjacency array, of size at least edgenbr (it can be more if the edge array is not compact).

The vendtab, velotab, vnlotab and vlbltab arrays are optional, and a NULL pointer can be passed as argument whenever they are not defined. Since, in Fortran, there is no null reference, passing the scotchfmeshbuild routine a reference equal to verttab in the velotab, vnlotab or vlbltab fields makes them be considered as missing arrays. Setting vendtab to refer to one cell after verttab yields the same result, as it is the exact semantics of a compact vertex array.

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

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, to call the SCOTCH_meshCheck routine on the newly created SCOTCH_Mesh structure, prior to any other calls to LIBSCOTCH routines.

Return values

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

8.12.3 SCOTCH_meshBuildElem

Synopsis

```
int SCOTCH_meshBuildElem (SCOTCH_Mesh *       meshptr,
                          const SCOTCH_Num    velmbas,
                          const SCOTCH_Num    vnodbas,
                          const SCOTCH_Num    velmnbr,
                          const SCOTCH_Num    vnodnbr,
                          const SCOTCH_Num *  verttab,
                          const SCOTCH_Num *  vendtab,
                          const SCOTCH_Num *  velotab,
                          const SCOTCH_Num *  vnlotab,
                          const SCOTCH_Num *  vlbltab,
                          const SCOTCH_Num    edgenbr,
                          const SCOTCH_Num *  edgetab)

scotchfmeshbuildelem (doubleprecision (*)  meshdat,
                      integer*num          velmbas,
                      integer*num          vnodbas,
                      integer*num          velmnbr,
                      integer*num          vnodnbr,
                      integer*num (*)      verttab,
                      integer*num (*)      vendtab,
                      integer*num (*)      velotab,
                      integer*num (*)      vnlotab,
                      integer*num (*)      vlbltab,
                      integer*num          edgenbr,
                      integer*num (*)      edgetab,
                      integer*num          ierr)
```

Description

The `SCOTCH_meshBuildElem` routine fills the source mesh structure pointed to by `meshptr` with an extension of the element adjacency data that is passed to it. `velmbas` and `vnodbas` are the base values for the input element and node vertices, respectively. However, in order to create a compact mesh, the element and node base values of the produced mesh will be equal to `velmbas` and (`velmbas` + `velmnbr`), respectively. `velmnbr` and `vnodnbr` are the number of element and node vertices, respectively. Typically, min(`velmbas`, `vnodbas`) is 0 for structures built from C and 1 for structures built from Fortran. `verttab` is the adjacency index array, of size (`velmnbr` + 1) if the edge array is compact (that is, if `vendtab` equals (`verttab` + 1) or `NULL`), or of size `velmnbr` else. `vendtab` is the adjacency end index array, of size `velmnbr` if it is disjoint from `verttab`. `velotab` is the element vertex load array, of size `velmnbr` if it exists. `vnlotab` is the node vertex load array, of size `vnodnbr` if it exists. `vlbltab` is the vertex label array, of size (`velmnbr` + `vnodnbr`) if it exists. `edgenbr` is the number of element-to-node arcs. `edgetab` is the element-to-node adjacency array, of size at least `edgenbr` (it can be more if the edge array is not compact).

The `vendtab`, `velotab`, `vnlotab` and `vlbltab` arrays are optional, and a `NULL` pointer can be passed as argument whenever they are not defined. Since, in Fortran, there is no null reference, passing the `scotchfmeshbuildelem` routine a reference equal to `verttab` in the `velotab`, `vnlotab` or `vlbltab` fields makes them be considered as missing arrays. Setting `vendtab` to refer

to one cell after verttab yields the same result, as it is the exact semantics of a compact vertex array.

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

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, to call the SCOTCH_meshCheck routine on the newly created SCOTCH_ Mesh structure, prior to any other calls to libScotch routines.

Return values

SCOTCH_meshBuildElem returns 0 if the mesh structure has been successfully computed from all of the input data, and 1 else.

8.12.4 SCOTCH_meshCheck

Synopsis

```
int SCOTCH_meshCheck (const SCOTCH_Mesh *  meshptr)

scotchfmeshcheck (doubleprecision (*)  meshdat,
                  integer              ierr)
```

Description

The SCOTCH_meshCheck routine checks the consistency of the given SCOTCH_Mesh structure. It can be used in client applications to determine if a mesh that has been created from used-generated data by means of the SCOTCH_meshBuild routine is consistent, prior to calling any other routines of the libScotch library.

Return values

SCOTCH_meshCheck returns 0 if mesh data are consistent, and 1 else.

8.12.5 SCOTCH_meshData

Synopsis

```
void SCOTCH_meshData (const SCOTCH_Mesh *  meshptr,
                      SCOTCH_Num *         vebaptr,
                      SCOTCH_Num *         vnbaptr,
                      SCOTCH_Num *         velmptr,
                      SCOTCH_Num *         vnodptr,
                      SCOTCH_Num **        verttab,
                      SCOTCH_Num **        vendtab,
                      SCOTCH_Num **        velotab,
                      SCOTCH_Num **        vnlotab,
                      SCOTCH_Num **        vlbltab,
                      SCOTCH_Num *         edgeptr,
                      SCOTCH_Num **        edgetab,
                      SCOTCH_Num *         degrptr)

scotchfmeshdata (doubleprecision (*)  meshdat,
                 integer*num (*)      indxtab,
                 integer*num          velobas,
                 integer*num          vnlobas,
                 integer*num          velmnbr,
                 integer*num          vnodnbr,
                 integer*idx          vertidx,
                 integer*idx          vendidx,
                 integer*idx          veloidx,
                 integer*idx          vnloidx,
                 integer*idx          vlblidx,
                 integer*num          edgenbr,
                 integer*idx          edgeidx,
                 integer*num          degrmax)
```

Description

The SCOTCH_meshData routine is the dual of the SCOTCH_meshBuild routine. It is a multiple accessor that returns scalar values and array references.

vebaptr and vnbaptr are pointers to locations that will hold the mesh base value for elements and nodes, respectively (the minimum of these two values is typically 0 for structures built from C and 1 for structures built from Fortran). velmptr and vnodptr are pointers to locations that will hold the number of element and node vertices, respectively. verttab is the pointer to a location that will hold the reference to the adjacency index array, of size (∗**velmptr** + ∗**vnodptr** + 1) if the adjacency array is compact, or of size (∗**velmptr** + ∗**vnodptr**) else. vendtab is the pointer to a location that will hold the reference to the adjacency end index array, and is equal to **verttab**+1 if the adjacency array is compact. velotab and vnlotab are pointers to locations that will hold the reference to the element and node vertex load arrays, of sizes ∗velmptr and ∗vnodptr, respectively. vlbltab is the pointer to a location that will hold the reference to the vertex label array, of size (∗**velmptr** + ∗**vnodptr**). edgeptr is the pointer to a location that will hold the number of arcs (that is, twice the number of edges). edgetab is the pointer to a location that will hold the reference to the adjacency array, of size at least edgenbr. degrptr is the pointer to a location that will hold the maximum vertex degree computed across all element and 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.

Since there are no pointers in Fortran, a specific mechanism is used to allow users to access mesh arrays. The scotchfmeshdata 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 vertidx, the index of verttab, if they do not exist. For instance, if some base array myarray (1) is passed as parameter indxtab, then the first cell of array verttab will be accessible as myarray(vertidx). 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 one 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. 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 8.1.5 for more information on this issue.

8.12.6 SCOTCH_meshExit

Synopsis

```
void SCOTCH_meshExit (SCOTCH_Mesh *  meshptr)
scotchfmeshexit (doubleprecision (*)  meshdat)
```

Description

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

8.12.7 SCOTCH_meshGraph

Synopsis

```
int SCOTCH_meshGraph (const SCOTCH_Mesh *  meshptr,
                      SCOTCH_Graph *       grafptr)
scotchfmeshgraph (doubleprecision (*)  meshdat,
                  doubleprecision (*)  grafdat,
                  integer              ierr)
```

Description

The SCOTCH_meshGraph routine builds a graph from a mesh. It creates in the SCOTCH_Graph structure pointed to by grafptr a graph having as many vertices as there are nodes in the SCOTCH_Mesh structure pointed to by meshptr, and where there is an edge between any two graph vertices if and only if there exists in the mesh an element containing both of the associated nodes. Consequently, all of the elements of the mesh are turned into cliques in the resulting graph.

In order to save memory space as well as computation time, in the current implementation of SCOTCH_meshGraph, some mesh arrays are shared with the graph structure. Therefore, one should make sure that the graph must no longer be used after the mesh structure is freed. The graph structure can be freed before or after the mesh structure, but must not be used after the mesh structure is freed.

Return values

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

8.12.8 SCOTCH_meshGraphDual

Synopsis

```
int SCOTCH_meshGraphDual (const SCOTCH_Mesh *  meshptr,
                          SCOTCH_Graph *       grafptr,
                          const SCOTCH_Num     nocoval)

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

Description

The SCOTCH_meshGraphDual routine builds a dual graph (i.e., an element graph) from a mesh. It creates, in the SCOTCH_Graph structure pointed to by grafptr, a graph having as many vertices as there are elements in the SCOTCH_Mesh structure pointed to by meshptr, and such that there exists an edge between any two graph vertices if there are at least nocoval nodes in common between the two corresponding elements in the source mesh, or if an element shares all of its nodes, minus one, with another element.

Return values

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

8.12.9 SCOTCH_meshInit

Synopsis

```
int SCOTCH_meshInit (SCOTCH_Mesh *  meshptr)
```

```
scotchfmeshinit (doubleprecision (*)  meshdat,
                 integer              ierr)
```

Description

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

Return values

SCOTCH_meshInit returns 0 if the mesh structure has been successfully initialized, and 1 else.

8.12.10 SCOTCH_meshLoad

Synopsis

```
int SCOTCH_meshLoad (SCOTCH_Mesh *  meshptr,
                     FILE *         stream,
                     SCOTCH_Num     baseval)

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

Description

The SCOTCH_meshLoad routine fills the SCOTCH_Mesh structure pointed to by meshptr with the source mesh description available from stream stream in the Scotch mesh format (see section 6.2).

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.

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

8.12.11 SCOTCH_meshSave

Synopsis

```
int SCOTCH_meshSave (const SCOTCH_Mesh *  meshptr,
                     FILE *               stream)

scotchfmeshsave (doubleprecision (*)  meshdat,
                 integer              fildes,
                 integer              ierr)
```

Description

The SCOTCH_meshSave routine saves the contents of the SCOTCH_Mesh structure pointed to by meshptr to stream stream, in the Scotch mesh format (see section 6.2).

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

8.12.12 SCOTCH_meshSize

Synopsis

```
void SCOTCH_meshSize (const SCOTCH_Mesh *  meshptr,
                      SCOTCH_Num *         velmptr,
                      SCOTCH_Num *         vnodptr,
                      SCOTCH_Num *         edgeptr)

scotchfmeshsize (doubleprecision (*)  meshdat,
                 integer*num          velmnbr,
                 integer*num          vnodnbr,
                 integer*num          edgenbr)
```

Description

The SCOTCH_meshSize routine fills the three areas of type SCOTCH_Num pointed to by velmptr, vnodptr and edgeptr with the number of element vertices, node vertices and arcs (that is, twice the number of edges) of the given mesh pointed to by meshptr, respectively.

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_meshLoad routine, in order to allocate auxiliary arrays of proper sizes. If the whole structure of the mesh is wanted, function SCOTCH_meshData should be preferred.

8.12.13 SCOTCH_meshSizeof

Synopsis

```
int SCOTCH_meshSizeof (void)
scotchfmeshsizeof (integer  size )
```

Description

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

8.12.14 SCOTCH_meshStat

Synopsis

```
void SCOTCH_meshStat (const SCOTCH_Mesh *  meshptr,
                      SCOTCH_Num *         vnlominptr,
                      SCOTCH_Num *         vnlomaxptr,
                      SCOTCH_Num *         vnlosumptr,
                      double *             vnloavgptr,
                      double *             vnlodltptr,
                      SCOTCH_Num *         edegminptr,
                      SCOTCH_Num *         edegmaxptr,
                      double *             edegavgptr,
                      double *             edegdltptr,
                      SCOTCH_Num *         ndegminptr,
                      SCOTCH_Num *         ndegmaxptr,
                      double *             ndegavgptr,
                      double *             ndegdltptr)
scotchfmeshstat (doubleprecision (*)  meshdat,
                 integer*num          vnlomin,
                 integer*num          vnlomax,
                 integer*num          vnlosum,
                 doubleprecision      vnloavg,
                 doubleprecision      vnlodlt,
                 integer*num          edegmin,
                 integer*num          edegmax,
                 doubleprecision      edegavg,
                 doubleprecision      edegdlt,
                 integer*num          ndegmin,
                 integer*num          ndegmax,
                 doubleprecision      ndegavg,
                 doubleprecision      ndegdlt)
```

Description

The SCOTCH_meshStat routine produces some statistics regarding the mesh structure pointed to by meshptr. vnlomin, vnlomax, vnlosum, vnloavg and vnlodlt are the minimum node vertex load, the maximum node vertex load, the sum of all node vertex loads, the average node vertex load, and the variance of the node vertex loads, respectively. edegmin, edegmax, edegavg and edegdlt are the minimum element vertex degree, the maximum element vertex degree, the average element vertex degree, and the variance of the element vertex degrees, respectively. ndegmin, ndegmax, ndegavg and ndegdlt are the minimum node vertex degree, the maximum node vertex degree, the average node vertex degree, and the variance of the node vertex degrees, respectively.

8.13 High-level mesh ordering routines

This routine provides high-level functionality and frees the user from the burden of calling in sequence several of the low-level routines described afterward.

8.13.1 SCOTCH_meshOrder

Synopsis

```
int SCOTCH_meshOrder (const SCOTCH_Mesh *   meshptr,
                      const SCOTCH_Strat *  straptr,
                      SCOTCH_Num *          permtab,
                      SCOTCH_Num *          peritab,
                      SCOTCH_Num *          cblkptr,
                      SCOTCH_Num *          rangtab,
                      SCOTCH_Num *          treetab)

scotchfmeshorder (doubleprecision (*)  meshdat,
                  doubleprecision (*)  stradat,
                  integer*num (*)      permtab,
                  integer*num (*)      peritab,
                  integer*num          cblknbr,
                  integer*num (*)      rangtab,
                  integer*num (*)      treetab,
                  integer              ierr)
```

Description

The SCOTCH_meshOrder routine computes a block ordering of the unknowns of the symmetric sparse matrix the adjacency structure of which is represented by the elements that connect the nodes of the source mesh structure pointed to by meshptr, using the ordering strategy pointed to by stratptr, and returns ordering data in the scalar pointed to by cblkptr and the four arrays permtab, peritab, rangtab and treetab.

The permtab, peritab, rangtab and treetab arrays should have been previously allocated, of a size sufficient to hold as many SCOTCH_Num integers as there are node vertices in the source mesh, plus one in the case of rangtab.

Any of the 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 meshptr in these fields will have the same effect.

On return, permtab holds the direct permutation of the unknowns, that is, node vertex i of the original mesh has index permtab[i] in the reordered mesh, while peritab holds the inverse permutation, that is, node vertex i in the reordered mesh had index peritab[i] in the original mesh. All of these indices are numbered according to the base value of the source mesh: permutation indices are numbered from min(**velmbas**, **vnodbas**) to **vnodnbr** + min(**velmbas**, **vnodbas**) − 1, that is, from 0 to **vnodnbr** − 1 if the mesh base is 0, and from 1 to **vnodnbr** if the mesh base is 1. The base value for mesh orderings is taken as min(**velmbas**, **vnodbas**), and not just as vnodbas, such that orderings that are computed on some mesh have exactly the same index range as orderings that would be computed on the graph obtained from the original mesh by means of the SCOTCH_meshGraph routine.

The three other result fields, *cblkptr, rangtab and treetab, contain data related to the block structure. *cblkptr holds the number of column blocks of the produced ordering, and rangtab holds the starting indices of each of the permuted column blocks, in increasing order, so that column block i starts at index rangtab[i] and ends at index (rangtab[i+1]−1), inclusive, in the new ordering. treetab holds the separators tree structure, that is, 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. Please refer to Section 8.2.5 for more information.

Return values

SCOTCH_meshOrder returns 0 if the ordering of the mesh has been successfully computed, and 1 else. In this last case, the rangtab, permtab, and peritab arrays may however have been partially or completely filled, but their contents are not significant.

8.14 Low-level mesh ordering routines

All of the following routines operate on a SCOTCH_Ordering structure that contains references to the permutation arrays to be filled during the mesh ordering process.

8.14.1 SCOTCH_meshOrderCheck

Synopsis

```
int SCOTCH_meshOrderCheck (const SCOTCH_Mesh *      meshptr,
                           const SCOTCH_Ordering *  ordeptr)

scotchfmeshordercheck (doubleprecision (*)  meshdat,
                       doubleprecision (*)  ordedat,
                       integer              ierr)
```

Description

The SCOTCH_meshOrderCheck routine checks the consistency of the given SCOTCH_Ordering structure pointed to by ordeptr.

Return values

SCOTCH_meshOrderCheck returns 0 if ordering data are consistent, and 1 else.

8.14.2 SCOTCH_meshOrderCompute

Synopsis

```
int SCOTCH_meshOrderCompute (const SCOTCH_Mesh *   meshptr,
                             SCOTCH_Ordering *     ordeptr,
                             const SCOTCH_Strat *  straptr)

scotchfmeshordercompute (doubleprecision (*)  meshdat,
                         doubleprecision (*)  ordedat,
                         doubleprecision (*)  stradat,
                         integer              ierr)
```

Description

The SCOTCH_meshOrderCompute routine computes a block ordering of the mesh structure pointed to by grafptr, using the mapping strategy pointed to by stratptr, and stores its result in the ordering structure pointed to by ordeptr.

On return, the ordering structure holds a block ordering of the given mesh (see section 8.14.4 for a description of the ordering fields).

Return values

SCOTCH_meshOrderCompute 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.

8.14.3 SCOTCH_meshOrderExit

Synopsis

```
void SCOTCH_meshOrderExit (const SCOTCH_Mesh *  meshptr,
                           SCOTCH_Ordering *    ordeptr)

scotchfmeshorderexit (doubleprecision (*)  meshdat,
                      doubleprecision (*)  ordedat)
```

Description

The SCOTCH_meshOrderExit function frees the contents of a SCOTCH_Ordering structure previously initialized by SCOTCH_meshOrderInit. All subsequent calls to SCOTCH_meshOrder* routines other than SCOTCH_meshOrderInit, using this structure as parameter, may yield unpredictable results.

8.14.4 SCOTCH_meshOrderInit

Synopsis

```
int SCOTCH_meshOrderInit (const SCOTCH_Mesh *  meshptr,
                          SCOTCH_Ordering *    ordeptr,
                          SCOTCH_Num *         permtab,
                          SCOTCH_Num *         peritab,
                          SCOTCH_Num *         cblkptr,
                          SCOTCH_Num *         rangtab,
                          SCOTCH_Num *         treetab)

scotchfmeshorderinit (doubleprecision (*)  meshdat,
                      doubleprecision (*)  ordedat,
                      integer*num (*)      permtab,
                      integer*num (*)      peritab,
                      integer*num          cblknbr,
                      integer*num (*)      rangtab,
                      integer*num (*)      treetab,
                      integer              ierr)
```

Description

The SCOTCH_meshOrderInit routine fills the ordering structure pointed to by ordeptr with all of the data that are passed to it. Thus, all subsequent calls to ordering routines such as SCOTCH_meshOrderCompute, using this ordering structure as parameter, will place ordering results in fields permtab, peritab, *cblkptr, rangtab or treetab, if they are not set to NULL.

permtab is the ordering permutation array, of size **vnodnbr**, peritab is the inverse ordering permutation array, of size **vnodnbr**, 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 **vnodnbr**+1, and treetab is the array holding the structure of the separators tree, of size **vnodnbr**. See the above manual page of SCOTCH_meshOrder, as well as section 8.2.5, for an explanation of the semantics of all of these fields.

The SCOTCH_meshOrderInit routine should be the first function to be called upon a SCOTCH_Ordering structure for ordering meshes. When the ordering structure is no longer of use, the SCOTCH_meshOrderExit function must be called, in order to to free its internal structures.

Return values

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

8.14.5 SCOTCH_meshOrderSave

Synopsis

```
int SCOTCH_meshOrderSave (const SCOTCH_Mesh *      meshptr,
                          const SCOTCH_Ordering *  ordeptr,
                          FILE *                   stream)

scotchfmeshordersave (doubleprecision (*)  meshdat,
                      doubleprecision (*)  ordedat,
                      integer              fildes,
                      integer              ierr)
```

Description

The SCOTCH_meshOrderSave routine saves the contents of the SCOTCH_Ordering structure pointed to by ordeptr to stream stream, in the Scotch ordering format (see section 6.6).

Return values

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

8.14.6 SCOTCH_meshOrderSaveMap

Synopsis

```
int SCOTCH_meshOrderSaveMap (const SCOTCH_Mesh *      meshptr,
                             const SCOTCH_Ordering *  ordeptr,
                             FILE *                   stream)

scotchfmeshordersavemap (doubleprecision (*)  meshdat,
                         doubleprecision (*)  ordedat,
                         integer              fildes,
                         integer              ierr)
```

Description

The SCOTCH_meshOrderSaveMap routine saves the block partitioning data associated with the SCOTCH_Ordering structure pointed to by ordeptr to stream stream, in the Scotch mapping format (see section 6.5). 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.

This mapping file can then be used by the gout program (see section 7.4.12) to produce pictures showing the different separators and blocks. Since gout only takes graphs as input, the mesh has to be converted into a graph by means of the gmk_msh program (see section 7.4.8).

Return values

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

8.14.7 SCOTCH_meshOrderSaveTree

Synopsis

```
int SCOTCH_meshOrderSaveTree (const SCOTCH_Mesh *      meshptr,
                              const SCOTCH_Ordering *  ordeptr,
                              FILE *                   stream)

scotchfmeshordersavetree (doubleprecision (*)  meshdat,
                          doubleprecision (*)  ordedat,
                          integer              fildes,
                          integer              ierr)
```

Description

The SCOTCH_meshOrderSaveTree routine saves the tree hierarchy information associated with the SCOTCH_Ordering structure pointed to by ordeptr to stream stream.

The format of the tree output file resembles the one of a mapping or ordering file: it is made up of as many lines as there are node vertices in the ordering. Each of these lines holds two integer numbers. The first one is the index or the label of the node vertex, starting from baseval, 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.

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 tree mapping file.

Return values

SCOTCH_meshOrderSaveTree returns 0 if the separators tree structure has been successfully written to stream, and 1 else.

8.15 Strategy handling routines

8.15.1 SCOTCH_stratAlloc

Synopsis

```
SCOTCH_Strat * SCOTCH_stratAlloc (void)
```

Description

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

Return values

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

8.15.2 SCOTCH_stratExit

Synopsis

```
void SCOTCH_stratExit (SCOTCH_Strat *  straptr)
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.

8.15.3 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.

8.15.4 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.

8.15.5 SCOTCH_stratSizeof

Synopsis

```
int SCOTCH_stratSizeof (void)
scotchfstratsizeof (integer  size )
```

Description

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

8.16 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.

8.16.1 SCOTCH_stratGraphBipart

Synopsis

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

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

Description

The SCOTCH_stratGraphBipart routine fills the strategy structure pointed to by straptr with the graph bipartitioning strategy string pointed to by string. From this point, the strategy structure can only be used as a graph bipartitioning strategy, to be used by function SCOTCH_archBuild, for instance.

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

Return values

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

8.16.2 SCOTCH_stratGraphClusterBuild

Synopsis

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

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

Description

The SCOTCH_stratGraphClusterBuild 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. 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 8.3.1 for a description of the available flags.

Return values

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

8.16.3 SCOTCH_stratGraphMap

Synopsis

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

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

Description

The SCOTCH_stratGraphMap routine fills the strategy structure pointed to by straptr with the graph mapping strategy string pointed to by string. From this point, the strategy structure can only be used as a mapping strategy, to be used by function SCOTCH_graphMap, for instance.

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

Return values

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

8.16.4 SCOTCH_stratGraphMapBuild

Synopsis

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

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

Description

The SCOTCH_stratGraphMapBuild 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. From this point, the strategy structure can only be used as a mapping strategy, to be used by function SCOTCH_

graphMap, for instance. See Section 8.3.1 for a description of the available flags.

Return values

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

8.16.5 SCOTCH_stratGraphPartOvl

Synopsis

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

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

Description

The SCOTCH_stratGraphPartOvl routine fills the strategy structure pointed to by straptr with the graph partitioning with overlap strategy string pointed to by string. From this point, the strategy structure can only be used as a partitioning with overlap strategy, to be used by function SCOTCH_graphPartOvl only.

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

Return values

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

8.16.6 SCOTCH_stratGraphPartOvlBuild

Synopsis

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

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

Description

The SCOTCH_stratGraphPartOvlBuild routine fills the strategy structure pointed to by straptr with a default partitioning with overlap strategy tuned according to the preference flags passed as flagval and to the desired number of parts partnbr and imbalance ratio balrat. From this point, the strategy structure can only be used as a partitioning with overlap strategy, to be used by function SCOTCH_graphPartOvl only. See Section 8.3.1 for a description of the available flags.

Return values

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

8.16.7 SCOTCH_stratGraphOrder

Synopsis

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

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

Description

The SCOTCH_stratGraphOrder routine fills the strategy structure pointed to by straptr with the graph ordering strategy string pointed to by string. From this point, the strategy structure can only be used as a graph ordering strategy, to be used by function SCOTCH_graphOrder, for instance.

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

Return values

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

8.16.8 SCOTCH_stratGraphOrderBuild

Synopsis

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

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

Description

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

See Section 8.3.1 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_stratGraphOrderBuild returns 0 if the strategy string has been successfully set, and 1 else.

8.16.9 SCOTCH_stratMeshOrder

Synopsis

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

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

Description

The SCOTCH_stratMeshOrder routine fills the strategy structure pointed to by straptr with the mesh ordering strategy string pointed to by string. From this point, strategy strat can only be used as a mesh ordering strategy, to be used by function SCOTCH_meshOrder, for instance.

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

Return values

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

8.16.10 SCOTCH_stratMeshOrderBuild

Synopsis

```
int SCOTCH_stratMeshOrderBuild (SCOTCH_Strat *    straptr,
                                const SCOTCH_Num  flagval,
                                const double      balrat)
```

```
scotchfstratmeshorderbuild (doubleprecision (*)  stradat,
                            integer*num          flagval,
                            doubleprecision      balrat,
                            integer              ierr)
```

Description

The SCOTCH_stratMeshOrderBuild routine fills the strategy structure pointed to by straptr with a default ordering strategy tuned according to the preference flags passed as flagval and to the desired nested dissection imbalance ratio balrat. From this point, the strategy structure can only be used as an ordering strategy, to be used by function SCOTCH_mesh Order, for instance. See Section 8.3.1 for a description of the available flags.

Return values

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

8.17 Geometry handling routines

Since the Scotch project is based on algorithms that rely on topology data only, geometry data do not play an important role in the libScotch library. They are only relevant to programs that display graphs, such as the gout program. However, since all routines that are used by the programs of the Scotch distributions have an interface in the libScotch library, there exist geometry handling routines in it, which manipulate SCOTCH_Geom structures.

Apart from the routines that create, destroy or access SCOTCH_Geom structures, all of the routines in this section are input/output routines, which read or write both SCOTCH_Graph and SCOTCH_Geom structures. We have chosen to define the interface of the geometry-handling routines such that they also handle graph or mesh topology because some external file formats mix these data, and that we wanted our routines to be able to read their data on the fly from streams that can only be read once, such as communication pipes. Having both aspects taken into account in a single call makes the writing of file conversion tools, such as gcv and mcv, very easy. When the file format from which to read or into which to write mixes both sorts of data, the geometry file pointer can be set to NULL, as it will not be used.

8.17.1 SCOTCH_geomAlloc

Synopsis

```
SCOTCH_Geom * SCOTCH_geomAlloc (void)
```

Description

The SCOTCH_geomAlloc function allocates a memory area of a size sufficient to store a SCOTCH_Geom structure. It is the user's responsibility to free this memory when it is no longer needed, using the SCOTCH_memFree routine.

The allocated space must be initialized before use, by means of the SCOTCH_geomInit routine.

Return values

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

8.17.2 SCOTCH_geomData

Synopsis

```
void SCOTCH_geomData (const SCOTCH_Geom *  geomptr,
                      SCOTCH_Num *         dimnptr,
                      double **            geomtab)

scotchfgeomdata (doubleprecision (*)  geomdat,
                 doubleprecision (*)  indxtab,
                 integer*num          dimnnbr,
                 integer*idx          geomidx)
```

Description

The SCOTCH_geomData routine is a multiple accessor to the contents of SCOTCH_Geom structures.

dimnptr is the pointer to a location that will hold the number of dimensions of the graph vertex or mesh node vertex coordinates, and will therefore be equal to 1, 2 or 3. geomtab is the pointer to a location that will hold the reference to the geometry coordinates, as defined in section 8.2.4.

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 the coordinate array. The scotchfgeomdata 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 a reference, the routine returns an integer, which represents the starting index of the coordinate array with respect to the base input array. For instance, if some base array myarray(1) is passed as parameter indxtab, then the first cell of array geomtab will be accessible as myarray (geomidx). In order for this feature to behave properly, the indxtab array must be double-precision-aligned with the geometry array. This is automatically enforced on most systems, but some care should be taken on systems that allow one to access data that is not double-aligned. On such systems, declaring the array after a dummy doubleprecision array can coerce the compiler into enforcing the proper alignment. 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 8.1.5 for more information on this issue.

8.17.3 SCOTCH_geomExit

Synopsis

```
void SCOTCH_geomExit (SCOTCH_Geom *  geomptr)
scotchfgeomexit (doubleprecision (*)  geomdat)
```

Description

The SCOTCH_geomExit function frees the contents of a SCOTCH_Geom structure previously initialized by SCOTCH_geomInit. All subsequent calls to SCOTCH_*Geom* routines other than SCOTCH_geomInit, using this structure as parameter, may yield unpredictable results.

8.17.4 SCOTCH_geomInit

Synopsis

```
int SCOTCH_geomInit (SCOTCH_Geom *  geomptr)
scotchfgeominit (doubleprecision (*)  geomdat,
                 integer              ierr)
```

Description

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

Return values

SCOTCH_geomInit returns 0 if the geometrical structure has been successfully initialized, and 1 else.

8.17.5 SCOTCH_geomSizeof

Synopsis

```
int SCOTCH_geomSizeof (void)
scotchfgeomsizeof (integer  size )
```

Description

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

8.17.6 SCOTCH_graphGeomLoadChac

Synopsis

```
int SCOTCH_graphGeomLoadChac (SCOTCH_Graph *  grafptr,
                              SCOTCH_Geom *   geomptr,
                              FILE *          grafstream,
                              FILE *          geomstream,
                              const char *    string)

scotchfgraphgeomloadchac (doubleprecision (*)  grafdat,
                          doubleprecision (*)  geomdat,
                          integer              graffildes,
                          integer              geomfildes,
                          character (*)        string)
```

Description

The SCOTCH_graphGeomLoadChac routine fills the SCOTCH_Graph structure pointed to by grafptr with the source graph description available from stream grafstream in the Chaco graph format [26]. Since this graph format does not handle geometry data, the geomptr and geomstream fields are not used, as well as the string field.

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

Return values

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

8.17.7 SCOTCH_graphGeomLoadHabo

Synopsis

```
int SCOTCH_graphGeomLoadHabo (SCOTCH_Graph *  grafptr,
                              SCOTCH_Geom *   geomptr,
                              FILE *          grafstream,
                              FILE *          geomstream,
                              const char *    string)

scotchfgraphgeomloadhabo (doubleprecision (*)  grafdat,
                          doubleprecision (*)  geomdat,
                          integer              graffildes,
                          integer              geomfildes,
                          character (*)        string)
```

Description

The SCOTCH_graphGeomLoadHabo routine fills the SCOTCH_Graph structure pointed to by grafptr with the source graph description available from stream grafstream in the Harwell-Boeing square assembled matrix format [11]. Since this graph format does not handle geometry data, the geomptr and geomstream fields are not used. Since multiple graph structures can be encoded sequentially within the same file, the string field contains the string representation of an integer number that codes the rank of the graph to read within the Harwell-Boeing file. It is equal to "0" in most cases.

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

Return values

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

8.17.8 SCOTCH_graphGeomLoadScot

Synopsis

```
int SCOTCH_graphGeomLoadScot (SCOTCH_Graph *  grafptr,
                              SCOTCH_Geom *   geomptr,
                              FILE *          grafstream,
                              FILE *          geomstream,
                              const char *    string)

scotchfgraphgeomloadscot (doubleprecision (*)  grafdat,
                          doubleprecision (*)  geomdat,
                          integer              graffildes,
                          integer              geomfildes,
                          character (*)        string)
```

Description

The SCOTCH_graphGeomLoadScot routine fills the SCOTCH_Graph and SCOTCH_Geom structures pointed to by grafptr and geomptr with the source graph description and geometry data available from streams graf stream and geomstream in the Scotch graph and geometry formats (see sections 6.1 and 6.3, respectively). The string field is not used.

Fortran users must use the PXFFILENO or FNUM functions to obtain the numbers of the Unix file descriptors graffildes and geomfildes associated with the logical units of the graph and geometry files.

Return values

SCOTCH_graphGeomLoadScot returns 0 if the graph topology and geometry have been successfully allocated and filled with the data read, and 1 else.

8.17.9 SCOTCH_graphGeomSaveChac

Synopsis

```
int SCOTCH_graphGeomSaveChac (const SCOTCH_Graph *  grafptr,
                              const SCOTCH_Geom *   geomptr,
                              FILE *                grafstream,
                              FILE *                geomstream,
                              const char *          string)

scotchfgraphgeomsavechac (doubleprecision (*)  grafdat,
                          doubleprecision (*)  geomdat,
                          integer              graffildes,
                          integer              geomfildes,
                          character (*)        string)
```

Description

The SCOTCH_graphGeomSaveChac routine saves the contents of the SCOTCH_Graph structure pointed to by grafptr to stream grafstream, in the Chaco graph format [26]. Since this graph format does not handle geometry data, the geomptr and geomstream fields are not used, as well as the string field.

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

Return values

SCOTCH_graphGeomSaveChac returns 0 if the graph structure has been successfully written to grafstream, and 1 else.

8.17.10 SCOTCH_graphGeomSaveScot

Synopsis

```
int SCOTCH_graphGeomSaveScot (const SCOTCH_Graph *  grafptr,
                              const SCOTCH_Geom *   geomptr,
                              FILE *                grafstream,
                              FILE *                geomstream,
                              const char *          string)

scotchfgraphgeomsavescot (doubleprecision (*)  grafdat,
                          doubleprecision (*)  geomdat,
                          integer              graffildes,
                          integer              geomfildes,
                          character (*)        string)
```

Description

The SCOTCH_graphGeomSaveScot routine saves the contents of the SCOTCH_Graph and SCOTCH_Geom structures pointed to by grafptr and geomptr to streams grafstream and geomstream, in the Scotch graph and geometry formats (see sections 6.1 and 6.3, respectively). The string field is not used.

Fortran users must use the PXFFILENO or FNUM functions to obtain the numbers of the Unix file descriptors graffildes and geomfildes associated with the logical units of the graph and geometry files.

Return values

SCOTCH_graphGeomSaveScot returns 0 if the graph topology and geometry have been successfully written to grafstream and geomstream, and 1 else.

8.17.11 SCOTCH_meshGeomLoadHabo

Synopsis

```
int SCOTCH_meshGeomLoadHabo (SCOTCH_Mesh *  meshptr,
                             SCOTCH_Geom *  geomptr,
                             FILE *         meshstream,
                             FILE *         geomstream,
                             const char *   string)

scotchfmeshgeomloadhabo (doubleprecision (*)  meshdat,
                         doubleprecision (*)  geomdat,
                         integer              meshfildes,
                         integer              geomfildes,
                         character (*)        string)
```

Description

The SCOTCH_meshGeomLoadHabo routine fills the SCOTCH_Mesh structure pointed to by meshptr with the source mesh description available from stream meshstream in the Harwell-Boeing square elemental matrix format [11]. Since this mesh format does not handle geometry data, the geomptr and geomstream fields are not used. Since multiple mesh structures can be encoded sequentially within the same file, the string field contains the string representation of an integer number that codes the rank of the mesh to read within the Harwell-Boeing file. It is equal to "0" in most cases.

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

Return values

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

8.17.12 SCOTCH_meshGeomLoadScot

Synopsis

```
int SCOTCH_meshGeomLoadScot (SCOTCH_Mesh *  meshptr,
                             SCOTCH_Geom *  geomptr,
                             FILE *         meshstream,
                             FILE *         geomstream,
                             const char *   string)

scotchfmeshgeomloadscot (doubleprecision (*)  meshdat,
                         doubleprecision (*)  geomdat,
                         integer              meshfildes,
                         integer              geomfildes,
                         character (*)        string)
```

Description

The SCOTCH_meshGeomLoadScot routine fills the SCOTCH_Mesh and SCOTCH_Geom structures pointed to by meshptr and geomptr with the source mesh description and node geometry data available from streams meshstream and geomstream in the Scotch mesh and geometry formats (see sections 6.2 and 6.3, respectively). The string field is not used.

Fortran users must use the PXFFILENO or FNUM functions to obtain the numbers of the Unix file descriptors meshfildes and geomfildes associated with the logical units of the mesh and geometry files.

Return values

SCOTCH_meshGeomLoadScot returns 0 if the mesh topology and node geometry have been successfully allocated and filled with the data read, and 1 else.

8.17.13 SCOTCH_meshGeomSaveScot

Synopsis

```
int SCOTCH_meshGeomSaveScot (const SCOTCH_Mesh *  meshptr,
                             const SCOTCH_Geom *  geomptr,
                             FILE *               meshstream,
                             FILE *               geomstream,
                             const char *         string)

scotchfmeshgeomsavescot (doubleprecision (*)  meshdat,
                         doubleprecision (*)  geomdat,
                         integer              meshfildes,
                         integer              geomfildes,
                         character (*)        string)
```

Description

The SCOTCH_meshGeomSaveScot routine saves the contents of the SCOTCH_Mesh and SCOTCH_Geom structures pointed to by meshptr and geomptr to streams meshstream and geomstream, in the Scotch mesh and geometry formats (see sections 6.2 and 6.3, respectively). The string field is not used.

Fortran users must use the PXFFILENO or FNUM functions to obtain the numbers of the Unix file descriptors meshfildes and geomfildes associated with the logical units of the mesh and geometry files.

Return values

SCOTCH_meshGeomSaveScot returns 0 if the mesh topology and node geometry have been successfully written to meshstream and geomstream, and 1 else.

8.18 Other data structure handling routines

8.18.1 SCOTCH_mapAlloc

Synopsis

```
SCOTCH_Mapping * SCOTCH_mapAlloc (void)
```

Description

The SCOTCH_mapAlloc function allocates a memory area of a size sufficient to store a SCOTCH_Mapping structure. It is the user's responsibility to free this memory when it is no longer needed, using the SCOTCH_memFree routine.

Return values

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

8.18.2 SCOTCH_mapSizeof

Synopsis

```
int SCOTCH_mapSizeof (void)
scotchfmapsizeof (integer  size )
```

Description

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

8.18.3 SCOTCH_orderAlloc

Synopsis

```
SCOTCH_Ordering * SCOTCH_orderAlloc (void)
```

Description

The SCOTCH_orderAlloc function allocates a memory area of a size sufficient to store a SCOTCH_Ordering structure. It is the user's responsibility to free this memory when it is no longer needed, using the SCOTCH_memFree routine.

Return values

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

8.18.4 SCOTCH_orderSizeof

Synopsis

```
int SCOTCH_orderSizeof (void)
scotchfordersizeof (integer  size )
```

Description

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

8.19 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 libscotcherr.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 -lscotcherr to the list of arguments of the linker, after the -lscotch argument.

8.19.1 SCOTCH_errorPrint

Synopsis

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

Description

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

8.19.2 SCOTCH_errorPrintW

Synopsis

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

Description

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

8.19.3 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.

8.20 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 8.1.6).

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 8.21).

8.20.1 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.

8.20.2 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.

8.20.3 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.

8.20.4 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; randmax[.

8.21 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, using default, compile-time settings. 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, a private, independent pseudo-random number generator, and a set of flags and option values defined at compile-time.

In order to associate a context with a task to be performed, one has to bind the context to a Scotch object (that is, a graph, a mesh or a distributed graph), so as to create a container of the same type as the object (that is, a graph, mesh or distributed graph, respectively). This dummy object, when passed to the libScotch routines, will allow them to retrieve the context information as well as a reference to the original object (see Figure 23). Several contexts can be bound to the same Scotch object, allowing different libScotch routines to work concurrently on it. If the same context is shared by several containers, these should never be used concurrently. The original object and the context of a container must never be freed before it.

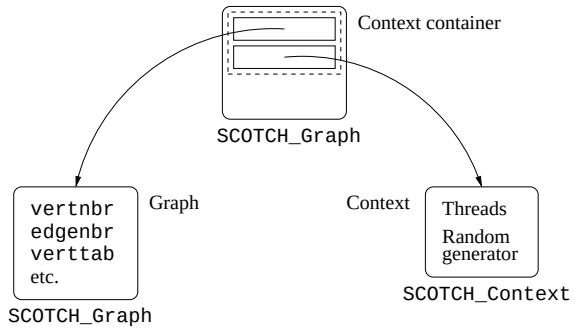


Figure 23: Dependencies between a context container and its underlying object and context. The context container is always seen as a Scotch object of the same type as the original object, for instance a SCOTCH_Graph in this example.

8.21.1 SCOTCH_contextInit

Synopsis

```
int SCOTCH_contextInit (SCOTCH_Context *  contptr)

scotchfcontextinit (doubleprecision (*)  contdat,
                    integer              ierr)
```

Description

The SCOTCH_contextInit function initializes a SCOTCH_Context structure so as to make it suitable for future operations. It should be the first function to be called upon a SCOTCH_Context structure. In the case when the context will be used to capture a pool of existing threads, it must be called only by the master thread.

When the context data is no longer of use, call function SCOTCH_contextExit to free its internal structures. A context must not be freed before the context containers that are created from it.

Return values

SCOTCH_contextInit returns 0 if the context structure has been successfully initialized, and 1 else.

8.21.2 SCOTCH_contextExit

Synopsis

```
void SCOTCH_contextExit (SCOTCH_Context *  contptr)
scotchfcontextexit (doubleprecision (*)  contdat)
```

Description

The SCOTCH_contextExit function frees the contents of a SCOTCH_Context structure previously initialized by SCOTCH_contextInit. In particular, it destroys the pseudo-random generator embedded in the context, and disposes of its threads. If the threads were created within the context, they are destroyed; if they were captured from an outside pool of threads (see SCOTCH_contextImport1), they are released.

8.21.3 SCOTCH_contextOptionGetNum

Synopsis

```
int SCOTCH_contextOptionGetNum (SCOTCH_Context *  contptr,
                                int               optinum)
                                SCOTCH_Num *      optivalptr)
scotchfcontextoptiongetnum (doubleprecision (*)  contdat,
                            integer              optinum,
                            integer*num          optival,
                            integer              ierr)
```

Description

The SCOTCH_contextOptionGetNum function retrieves, in the SCOTCH_Num value optival pointed to by optivalptr, the current integer context value of index optinum. Integer option index values are of the form SCOTCH_OPTIONNUM*, and range from 0 to (SCOTCH_OPTIONNUMNBR - 1), inclusive. This allows users to capture in an array of SCOTCH_Nums the values of relevant integer execution options, for the sake of reproducibility.

Return values

SCOTCH_contextOptionGetNum returns 0 if the value has been properly retrieved, and 1 in case of an invalid optinum index.

8.21.4 SCOTCH_contextOptionSetNum

Synopsis

```
int SCOTCH_contextOptionSetNum (SCOTCH_Context *  contptr,
                                int               optinum)
                                SCOTCH_Num        optival)

scotchfcontextoptionsetnum (doubleprecision (*)  contdat,
                            integer              optinum,
                            integer*num          optival,
                            integer              ierr)
```

Description

The SCOTCH_contextOptionSetNum function sets the integer context value of index optinum with the value optival.

Available integer option values are the following:

- SCOTCH_OPTIONNUMDETERMINISTIC: a value of 0 induces a non-deterministic behavior (*i.e.*, different runs may yield different results), while a value of 1 induces a deterministic behavior across multiple runs, for any number of threads (yet, for the same number of MPI processes, as data distribution varies across numbers of MPI processes). The initial value of this option at run time is defined by the compilation option SCOTCH_DETERMINISTIC (see the SCOTCH installation instructions). A deterministic behavior implies the use of a fixed random seed (see below).
- SCOTCH_OPTIONNUMFIXEDSEED: a value of 0 induces that a distinct random seed be selected for each launch of a program using the LIBSCOTCH library, while a value of 1 induces that the same, fixed random seed be used at each launch. The initial value of this option at run time is defined by the compilation option COMMON_RANDOM_FIXED_SEED (see the SCOTCH installation instructions).

Return values

SCOTCH_contextOptionSetNum returns 0 if the value has been properly set, and 1 on error.

8.21.5 SCOTCH_contextRandomClone

Synopsis

```
int SCOTCH_contextRandomClone (SCOTCH_Context *  contptr)

scotchfcontextrandomclone (doubleprecision (*)  contdat,
                           integer              ierr)
```

Description

The SCOTCH_contextRandomClone routine clones into the given context the current state of the global pseudo-random generator. It allows the user to run concurrently several routines of the LIBSCOTCH library in a reproducibile way, since the cloned generators within two different contexts can be parametrized independently.

Return values

SCOTCH_contextRandomClone returns 0 if the pseudo-random generator has been successfully cloned, and 1 else.

8.21.6 SCOTCH_contextRandomReset

Synopsis

```
void SCOTCH_contextRandomReset (SCOTCH_Context *  contptr)
scotchfcontextrandomreset (doubleprecision (*)  contdat)
```

Description

The SCOTCH_contextRandomReset routine resets the seed of the pseudo-random generator used by the given context. This generator may either be the default global pseudo-random generator of the LIBSCOTCH library (the routine then performs like SCOTCH_randomReset) or a private generator created by means of the SCOTCH_contextRandomClone function.

Please see the manual page of SCOTCH_randomReset, page 171, for more information on the operations of the pseudo-random generator resetting routines.

8.21.7 SCOTCH_contextRandomSeed

Synopsis

```
void SCOTCH_contextRandomSeed (SCOTCH_Context *  contptr,
                               SCOTCH_Num        seedval)
scotchfcontextrandomseed (doubleprecision (*)  contdat,
                          integer*num          seedval)
```

Description

The SCOTCH_contextRandomSeed routine sets to seedval the seed of the pseudo-random generator used by the given context. This generator may either be the default global pseudo-random generator of the LIBSCOTCH library (the routine then performs like SCOTCH_randomSeed) or a private generator created by means of the SCOTCH_contextRandomClone function.

Please see the manual page of SCOTCH_randomSeed, page 172, for more information on the operations of the pseudo-random generator seed setting routines.

8.21.8 SCOTCH_contextSizeof

Synopsis

```
int SCOTCH_contextSizeof (void)
scotchfcontextsizeof (integer  size )
```

Description

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

8.21.9 SCOTCH_contextThreadImport1

Synopsis

```
int SCOTCH_contextThreadImport1 (SCOTCH_Context *  contptr,
                                 int               thrdnbr)
scotchfcontextthreadimport1 (doubleprecision (*)  contdat,
                             integer              thrdnbr,
                             integer              ierr)
```

Description

The SCOTCH_contextThreadImport1 function initiates the capture, into the given context, of an existing pool of threads. It must be called only by the master thread, which provides thrdnbr, the overall number of threads to be included in the context (including the master thread).

See Listing 1, page 178 for a sample of multi-threaded code using SCOTCH_contextThreadImport1 and SCOTCH_contextThreadImport2 to capture a pool of existing threads and use it for libScotch computations.

Return values

SCOTCH_contextThreadImport1 returns 0 if the context thread structure has been successfully initialized, and 1 else.

8.21.10 SCOTCH_contextThreadImport2

Synopsis

```
int SCOTCH_contextThreadImport2 (SCOTCH_Context *  contptr,
                                 int               thrdnum)
scotchfcontextthreadimport2 (doubleprecision (*)  contdat,
                             integer              thrdnum,
                             integer              ierr)
```

Listing 1: Sample code for capturing a pool of threads and launching multi-threaded computations using this pool. All LIBSCOTCH routines which are passed the context graph (instead of the regular graph) will make use of the captured threads.

```
1  SCOTCH_Context contdat; /* Context data */
2  SCOTCH_Graph graftab[2]; /* A regular graph and a container graph */
3  ...
4  if (thrdnum == 0) {
5    SCOTCH_graphInit (&graftab[0]); /* Initialize regular graph */
6    SCOTCH_graphLoad (&graftab[0], ...); /* User places data in graph
         structure */
7    ...
8    SCOTCH_contextInit (&contdat); /* Initialize context */
9    SCOTCH_contextRandomClone (&contdat); /* Set private random generator */
10   SCOTCH_contextThreadImport1 (&contdat, thrdnbr); /* Give number of
         threads */
11 }
12 user_thread_barrier (); /* User makes sure there is synchronization
       across threads here */
13 SCOTCH_contextThreadImport2 (&contdat, thrdnum); /* Every thread gives
       its rank */
14 ... /* From here, all slave threads are blocked until context is
       destroyed */
15 if (thrdnum == 0) {
16   SCOTCH_graphInit (&graftab[1]); /* Initialize container graph */
17   SCOTCH_contextBindGraph (&contdat, &graftab[0], &graftab[1]); /* Bind
         context */
18   ...
19   SCOTCH_graphOrder (&graftab[1], ...); /* Use container graph for multi-
         threading */
20   ...
21   SCOTCH_graphExit (&graftab[1]); /* Destroy container */
22   SCOTCH_contextExit (&contdat); /* Destroy context; slave threads are
         released */
23   ...
24   SCOTCH_graphExit (&graftab[0]); /* Destroy regular graph */
25 }
```

Description

The SCOTCH_contextThreadImport2 function finalizes the capture, into the given context, of an existing pool of threads. It must be called by all threads, including the master thread, after the latter has returned from SCOTCH_contextThreadImport1; hence, some synchronization (e.g., a barrier) must be implemented between the two routines, for all threads to wait until the master thread has completed setting-up the context structure.

Each thread provides thrdnum, its rank within the thread pool; it must be 0 for the master thread. On success, only the master thread returns immediatly from the routine; all slave threads are captured within the context and put to sleep. The master thread can then bind the context to a Scotch object (see SCOTCH_contextBindGraph and SCOTCH_contextBindMesh), and call libScotch routines using the created context container.

When the context is no longer of use, it can be destroyed by the master thread by calling SCOTCH_contextExit. All slave threads will then return from their call to SCOTCH_contextThreadImport2.

See Listing 1, page 178 for an example of code using this routine.

Return values

SCOTCH_contextThreadImport2 returns 1 to all threads in case of an error. Else, it returns 0 to the master thread immediately, and 0 to the slave threads once the capturing context has been destroyed by the master thread.

8.21.11 SCOTCH_contextThreadSpawn

Synopsis

```
int SCOTCH_contextThreadSpawn (SCOTCH_Context *  contptr,
                               int               thrdnbr,
                               int *             coretab)

scotchfcontextthreadspawn (doubleprecision (*)  contdat,
                           integer              thrdnbr,
                           integer              coretab,
                           integer              ierr)
```

Description

The SCOTCH_contextThreadSpawn function populates the given context contptr with (thrdnbr − 1) slave threads which, in addition to the master, calling thread, will enable thrdnbr threads to participate in the computation of the libScotch routines called using this context.

When coretab is not equal to NULL or &thrdnbr, it represents an array of thrdnbr integer values that represent the logical indices of the cores onto which the threads will be bound. Valid core indices range from 0 to the number of cores available on the platform, minus one. Core indices above this threshold are subject to modulus reduction. A core index of −1 results in

Listing 2: Sample code for running multi-threaded computations using a dedicated pool of threads. This code is run by the main thread. All LIBSCOTCH routines which are passed the context graph (instead of the regular graph) will make use of the dedicated pool. If LIBSCOTCH routines are called on the regular graph, a local pool is created and destroyed for every function call.

```
1  SCOTCH_Context contdat; /* Context data */
2  SCOTCH_Graph graftab[2]; /* A regular graph and a container graph */
3  ...
4  SCOTCH_graphInit (&graftab[0]); /* Initialize regular graph */
5  SCOTCH_graphLoad (&graftab[0], ...); /* User places data in graph
       structure */
6  ...
7  SCOTCH_contextInit (&contdat); /* Initialize context */
8  SCOTCH_contextThreadSpawn (&contdat, thrdnbr, NULL); /* Set number of
       threads */
9  ...
10 SCOTCH_graphInit (&graftab[1]); /* Initialize container graph */
11 SCOTCH_contextBindGraph (&contdat, &graftab[0], &graftab[1]); /* Bind
       context */
12 ...
13 SCOTCH_graphOrder (&graftab[1], ...); /* Use container graph for multi-
       threading */
14 ...
15 SCOTCH_graphExit (&graftab[1]); /* Destroy container */
16 SCOTCH_contextExit (&contdat); /* Destroy context */
17 ...
18 SCOTCH_graphExit (&graftab[0]); /* Destroy regular graph */
```

the thread not being bound to a specific core (which may be detrimental to memory locality if threads are not always executed on the same core). The conversion from logical core numbers to physical core numbers depends on the platform features selected at compile time, e.g., "-DCOMMON_PTHREAD_AFFINITY_LINUX" (which binds theads to logical cores without accounting for hardware locality) or "-DCOMMON_PTHREAD_AFFINITY_HWLOC". When no affinity module is defined at compile time, threads are left unbound.

See Listing 2, page 180 for a sample of sequential code using SCOTCH_contextThreadSpawn to create a thread pool of a prescribed number of threads.

Return values

SCOTCH_contextThreadSpawn returns 0 if the context thread structure has been successfully initialized, and 1 else.

8.21.12 SCOTCH_contextBindGraph

Synopsis

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

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

Description

The SCOTCH_contextBindGraph function initializes a context container object cntgraf, with a type compatible with a SCOTCH_Graph structure, to make it reference both the given genuine SCOTCH_Graph structure orggraf and the SCOTCH_Context structure cont. The context container can then be used by routines of the LIBSCOTCH library that expect a SCOTCH_Graph, 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_graphExit to free its internal structures. The original graph and the context can then also be freed by using the adequate routines.

Return values

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

8.21.13 SCOTCH_contextBindMesh

Synopsis

```
int SCOTCH_contextBindMesh (SCOTCH_Context *  contptr,
                            SCOTCH_Mesh *     orgmeshptr,
                            SCOTCH_Mesh *     cntmeshptr)

scotchfcontextBindMesh (doubleprecision (*)  contdat,
                        doubleprecision (*)  orgmeshdat,
                        doubleprecision (*)  cntmeshdat,
                        integer              ierr)
```

Description

The SCOTCH_contextBindMesh function initializes a context container object cntmesh, with a type compatible with a SCOTCH_Mesh structure, to make it reference both the given genuine SCOTCH_Mesh structure orgmesh and the SCOTCH_Context structure cont. The context container can then be used by routines of the LIBSCOTCH library that expect a SCOTCH_Mesh, 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_graphExit to free its internal structures. The original graph and the context can then also be freed by using the adequate routines.

Return values

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

8.22 Memory management

8.22.1 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 8.1.5 for more information.

8.22.2 SCOTCH_memFree

Synopsis

```
void SCOTCH_memFree (void *  dataptr)
```

Description

The SCOTCH_memFree routine frees the memory space allocated by routines such as SCOTCH_graphAlloc, SCOTCH_meshAlloc, or SCOTCH_strat Alloc.

The standard free routine of the libc must not be used for this purpose. Else, the allocated memory will not be considered as properly released by memory accounting routines SCOTCH_memCur and SCOTCH_memMax, and segmentation errors would happen when the COMMON_MEMORY_CHECK compile flag is set.

On the opposite, if the user has allocated memory by himself according to the size information provided by the SCOTCH_*Sizeof routines, it is his responsibility to free this memory using the corresponding memory freeing routine of his environment.

8.22.3 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 8.1.5 for more information.

8.23 Miscellaneous routines

8.23.1 SCOTCH_numSizeof

Synopsis

```
int SCOTCH_numSizeof (void)
scotchfnumsizeof (integer  size )
```

Description

The SCOTCH_numSizeof routine returns the size, in bytes, of a SCOTCH_Num. This information is useful to export the interface of the libSCOTCH to interpreted languages, without access to the "scotch.h" include file.

8.23.2 SCOTCH_version

Synopsis

```
int SCOTCH_version (int *  versptr,
                    int *  relaptr,
                    int *  patcptr)

scotchfversion (integer  versval,
                integer  relaval,
                integer  patcval)
```

Description

The SCOTCH_version routine writes the version, release and patchlevel numbers of the Scotch library that is currently being used, to integer values *versptr, *relaptr and patcptr, respectively. This routine is mainly useful for applications willing to record runtime information, such as the library against which they are dynamically linked.

8.24 MeTiS compatibility library

The MeTiS compatibility library provides stubs which redirect some calls to MeTiS routines to the corresponding Scotch counterparts. In order to use this feature, the only thing to do is to re-link the existing software with the libscotchmetis library, and eventually with the original MeTiS library if the software uses MeTiS routines which do not need to have Scotch equivalents, such as graph transformation routines. In that latter case, the "-lscotchmetis" argument must be placed before the "-lmetis" one (and of course before the "-lscotch" one too), so that routines that are redefined by Scotch are chosen instead of their MeTiS counterpart. When no other MeTiS routines than the ones redefined by Scotch are used, the "-lmetis" argument can be omitted. See Section 10 for an example.

8.24.1 METIS_EdgeND

Synopsis

```
void METIS_EdgeND (const SCOTCH_Num *  n,
                   const SCOTCH_Num *  xadj,
                   const SCOTCH_Num *  adjncy,
                   const SCOTCH_Num *  numflag,
                   const SCOTCH_Num *  options,
                   SCOTCH_Num *        perm,
                   SCOTCH_Num *        iperm)

metis_edgend (integer*num      n,
              integer*num (*)  xadj,
              integer*num (*)  adjncy,
              integer*num      numflag,
              integer*num (*)  options,
              integer*num (*)  perm,
              integer*num (*)  iperm)
```

Description

The METIS_EdgeND function performs a nested dissection ordering of the graph passed as arrays xadj and adjncy, using the default SCOTCH ordering strategy.

Conforming to the METIS API, the base value of the numbering can be defined by setting options[METIS_OPTION_NUMBERING]. It is the only option supported by SCOTCH. The perm and iperm arrays have the opposite meaning as in SCOTCH: the METIS perm array holds what is called "inverse permutation" in SCOTCH, while iperm holds what is called "direct permutation" in SCOTCH.

While SCOTCH has also both node and edge separation capabilities, all of the three METIS stubs METIS_EdgeND, METIS_NodeND and METIS_NodeWND call the same SCOTCH routine, which uses the SCOTCH default ordering strategy proved to be efficient in most cases.

8.24.2 METIS_MeshToDual

Synopsis

```
int METIS_MeshToDual (const SCOTCH_Num * const  ne,
                      const SCOTCH_Num * const  nn,
                      const SCOTCH_Num * const  eptr,
                      const SCOTCH_Num * const  eind,
                      const SCOTCH_Num * const  ncommon,
                      const SCOTCH_Num * const  numflag,
                      SCOTCH_Num ** const       xadj,
                      SCOTCH_Num ** const       adjncy)

metis_meshtodual (integer*num       ne,
                  integer*num       nn,
                  integer*num (*)   eptr,
                  integer*num (*)   eind,
                  integer*num       ncommon,
                  integer*num       numflag,
                  integer*num (**)  xadj,
                  integer*num (**)  adjncy)
```

Description

Given the eptr and eind arrays describing the element-to-node adjacency of a mesh with *ne elements and *nn nodes, the METIS_MeshToDual function computes the dual graph, *i.e.* the graph of elements, of this mesh. The vertices of the dual graph represent the elements of the mesh, and there exists an edge between any two graph vertices if and only if there are at least ncommon shared nodes between the two corresponding elements in the mesh. The numflag parameter corresponds to the base value of the numbering: 0 for C, and 1 for Fortran.

On return, the `xadj` and `adjncy` arrays are allocated, by way of the `malloc()` routine, and contain the adjacency list of the dual graph. It is the user's responsibility to free these arrays, using the `free()` routine, whenever these arrays are no longer necessary. If the function fails, `xadj` is set to `NULL`.

8.24.3 METIS_NodeND

Synopsis

```
void METIS_NodeND (const SCOTCH_Num *  n,
                   const SCOTCH_Num *  xadj,
                   const SCOTCH_Num *  adjncy,
                   const SCOTCH_Num *  numflag,
                   const SCOTCH_Num *  options,
                   SCOTCH_Num *        perm,
                   SCOTCH_Num *        iperm)

metis_nodend (integer*num      n,
              integer*num (*)  xadj,
              integer*num (*)  adjncy,
              integer*num      numflag,
              integer*num (*)  options,
              integer*num (*)  perm,
              integer*num (*)  iperm)
```

Description

The `METIS_NodeND` function performs a nested dissection ordering of the graph passed as arrays `xadj` and `adjncy`, using the default Scotch ordering strategy.

Conforming to the MeTiS API, the base value of the numbering can be defined by setting `options[METIS_OPTION_NUMBERING]`. It is the only option supported by Scotch. The `perm` and `iperm` arrays have the opposite meaning as in Scotch: the MeTiS `perm` array holds what is called "inverse permutation" in Scotch, while `iperm` holds what is called "direct permutation" in Scotch.

While Scotch has also both node and edge separation capabilities, all of the three MeTiS stubs `METIS_EdgeND`, `METIS_NodeND` and `METIS_NodeWND` call the same Scotch routine, which uses the Scotch default ordering strategy proved to be efficient in most cases.

8.24.4 METIS_NodeWND

Synopsis

```
void METIS_NodeWND (const SCOTCH_Num *  n,
                    const SCOTCH_Num *  xadj,
                    const SCOTCH_Num *  adjncy,
                    const SCOTCH_Num *  vwgt,
                    const SCOTCH_Num *  numflag,
                    const SCOTCH_Num *  options,
                    SCOTCH_Num *        perm,
                    SCOTCH_Num *        iperm)

metis_nodwend (integer*num      n,
               integer*num (*)  xadj,
               integer*num (*)  adjncy,
               integer*num (*)  vwgt,
               integer*num      numflag,
               integer*num (*)  options,
               integer*num (*)  perm,
               integer*num (*)  iperm)
```

Description

The METIS_NodeWND function performs a nested dissection ordering of the graph passed as arrays xadj, adjncy and vwgt, using the default SCOTCH ordering strategy.

Conforming to the METIS API, the base value of the numbering can be defined by setting options[METIS_OPTION_NUMBERING]. It is the only option supported by SCOTCH. The perm and iperm arrays have the opposite meaning as in SCOTCH: the METIS perm array holds what is called "inverse permutation" in SCOTCH, while iperm holds what is called "direct permutation" in SCOTCH.

While SCOTCH has also both node and edge separation capabilities, all of the three METIS stubs METIS_EdgeND, METIS_NodeND and METIS_NodeWND call the same SCOTCH routine, which uses the SCOTCH default ordering strategy proved to be efficient in most cases.

8.24.5 METIS_PartGraphKway

Synopsis

```
void METIS_PartGraphKway (const SCOTCH_Num *  n,
                          const SCOTCH_Num *  xadj,
                          const SCOTCH_Num *  adjncy,
                          const SCOTCH_Num *  vwgt,
                          const SCOTCH_Num *  adjwgt,
                          const SCOTCH_Num *  wgtflag,
                          const SCOTCH_Num *  numflag,
                          const SCOTCH_Num *  nparts,
                          const SCOTCH_Num *  options,
                          SCOTCH_Num *        edgecut,
                          SCOTCH_Num *        part)
```

```
metis_partgraphkway (integer*num      n,
                     integer*num (*)  xadj,
                     integer*num (*)  adjncy,
                     integer*num (*)  vwgt,
                     integer*num (*)  adjwgt,
                     integer*num      wgtflag,
                     integer*num      numflag,
                     integer*num      nparts,
                     integer*num (*)  options,
                     integer*num      edgecut,
                     integer*num (*)  part)
```

Description

The METIS_PartGraphKway function performs a mapping onto the complete graph of the graph represented by arrays xadj, adjncy, vwgt and adjwgt, using the default SCOTCH mapping strategy.

Conforming to the METIS API, the base value of the numbering can be defined by setting options[METIS_OPTION_NUMBERING]. It is the only option supported by SCOTCH. The part array has the same meaning as the parttab array of SCOTCH.

All of the three METIS stubs METIS_PartGraphKway, METIS_PartGraph Recursive and METIS_PartGraphVKway call the same SCOTCH routine, which uses the SCOTCH default mapping strategy proved to be efficient in most cases.

8.24.6 METIS_PartGraphRecursive

Synopsis

```
void METIS_PartGraphRecursive (const SCOTCH_Num *  n,
                               const SCOTCH_Num *  xadj,
                               const SCOTCH_Num *  adjncy,
                               const SCOTCH_Num *  vwgt,
                               const SCOTCH_Num *  adjwgt,
                               const SCOTCH_Num *  wgtflag,
                               const SCOTCH_Num *  numflag,
                               const SCOTCH_Num *  nparts,
                               const SCOTCH_Num *  options,
                               SCOTCH_Num *        edgecut,
                               SCOTCH_Num *        part)
```

```
metis_partgraphrecursive (integer*num      n,
                          integer*num (*)  xadj,
                          integer*num (*)  adjncy,
                          integer*num (*)  vwgt,
                          integer*num (*)  adjwgt,
                          integer*num      wgtflag,
                          integer*num      numflag,
                          integer*num      nparts,
                          integer*num (*)  options,
                          integer*num      edgecut,
                          integer*num (*)  part)
```

Description

The METIS_PartGraphRecursive function performs a mapping onto the complete graph of the graph represented by arrays xadj, adjncy, vwgt and adjwgt, using the default SCOTCH mapping strategy.

Conforming to the METIS API, the base value of the numbering can be defined by setting options[METIS_OPTION_NUMBERING]. It is the only option supported by SCOTCH. The part array has the same meaning as the parttab array of SCOTCH. To date, the computation of the edgecut field requires extra processing, which increases running time to a small extent.

All of the three METIS stubs METIS_PartGraphKway, METIS_PartGraph Recursive and METIS_PartGraphVKway call the same SCOTCH routine, which uses the SCOTCH default mapping strategy proved to be efficient in most cases.

8.24.7 METIS_PartGraphVKway

Synopsis

```
void METIS_PartGraphVKway (const SCOTCH_Num *  n,
                           const SCOTCH_Num *  xadj,
                           const SCOTCH_Num *  adjncy,
                           const SCOTCH_Num *  vwgt,
                           const SCOTCH_Num *  vsize,
                           const SCOTCH_Num *  wgtflag,
                           const SCOTCH_Num *  numflag,
                           const SCOTCH_Num *  nparts,
                           const SCOTCH_Num *  options,
                           SCOTCH_Num *        volume,
                           SCOTCH_Num *        part)
```

```
metis_partgraphvkway (integer*num      n,
                      integer*num (*)  xadj,
                      integer*num (*)  adjncy,
                      integer*num (*)  vwgt,
                      integer*num (*)  vsize,
                      integer*num      wgtflag,
                      integer*num      numflag,
                      integer*num      nparts,
                      integer*num (*)  options,
                      integer*num      volume,
                      integer*num (*)  part)
```

Description

The METIS_PartGraphVKway function performs a mapping onto the complete graph of the graph represented by arrays xadj, adjncy, vwgt and vsize, using the default SCOTCH mapping strategy.

Conforming to the METIS API, the base value of the numbering can be defined by setting options[METIS_OPTION_NUMBERING]. It is the only option supported by SCOTCH. The part array has the same meaning as the parttab array of SCOTCH.

Since SCOTCH does not have methods for explicitely reducing the communication volume according to the metric of METIS_PartGraphVKway, this routine creates a temporary edge weight array such that each edge (u, v) receives a weight equal to $mbox{\texttt{vsize}}(u) + mbox{\texttt{vsize}}(v)$. Consequently, edges which are incident to highly communicating vertices will be less likely to be cut. However, the communication volume value returned by this routine is exactly the one which would be returned by METIS with respect to the output partition. Users interested in minimizing the exact communication volume should consider using hypergraphs, implemented in SCOTCH as meshes (see Section 8.2.3).

All of the three METIS stubs METIS_PartGraphKway, METIS_PartGraph Recursive and METIS_PartGraphVKway call the same SCOTCH routine, which uses the SCOTCH default mapping strategy proved to be efficient in most cases.

8.24.8 METIS_PartMeshDual

Synopsis

```
int METIS_PartMeshDual (const SCOTCH_Num * const  ne,
                        const SCOTCH_Num * const  nn,
                        const SCOTCH_Num * const  eptr,
                        const SCOTCH_Num * const  eind,
                        const SCOTCH_Num * const  vwgt,
                        const SCOTCH_Num * const  vsize,
                        const SCOTCH_Num * const  ncommon,
                        const SCOTCH_Num * const  nparts,
                        const double * const      tpwgts,
                        const SCOTCH_Num * const  options,
                        SCOTCH_Num * const        objval,
                        SCOTCH_Num * const        epart,
                        SCOTCH_Num * const        npart)

metis_metistodual (integer*num           ne,
                   integer*num           nn,
                   integer*num (*)       eptr,
                   integer*num (*)       eind,
                   integer*num (*)       vwgt,
                   integer*num (*)       vsize,
                   integer*num           ncommon,
                   integer*num           nparts,
                   doubleprecision* (*)  tpwgts,
                   integer*num (*)       options,
                   integer*num           objval,
                   integer*num (*)       epart,
                   integer*num (*)       npart)
```

Description

Given the `eptr` and `eind` arrays describing the element-to-node adjacency of a mesh with `*ne` elements and `*nn` nodes, the `METIS_PartMeshDual` function computes a partitioning into `nparts` of the dual graph of this mesh. The vertices of the dual graph represent the elements of the mesh, and there exists an edge between any two graph vertices if and only if there are at least `ncommon` shared nodes between the two corresponding elements in the mesh.

Conforming to the MeTiS API, the base value of the numbering can be defined by setting `options[METIS_OPTION_NUMBERING]`. It is the only option supported by Scotch.

On return, the `epart` and `npart` arrays contain the element and node partitions, respectively. If the function fails, `*objval` is set to a negative value that corresponds to the relevant `METIS_ERROR_*` code.

8.24.9 METIS_SetDefaultOptions

Synopsis

```
int METIS_SetDefaultOptions (SCOTCH_Num * const  options)
metis_setdefaultoptions (integer*num (*)  options)
```

Description

This function partially fills the `options` array with the values that are relevant to the operations of the MeTiS compatibility library. Other values are set to 0.

The supported options to date are `METIS_OPTION_NUMBERING` (which allows one to set the `baseval` value of libScotch routines) and `METIS_OPTION_OBJTYPE` (only for computing the partition cost, as Scotch does not try directly to minimize communication volume).

9 Installation

Version 7.0 of the Scotch software package is distributed as free/libre software under the CeCILL-C free/libre software license [7], which is very similar to the GNU LGPL license. Therefore, it is no longer distributed as a set of binaries, but instead in the form of a source distribution, which can be downloaded from the Scotch 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.

9.1 Thread issues

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

9.2 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.

9.3 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 sections 8.7.9, 8.12.5 and 8.17.2). 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.

10 Examples

This section contains chosen examples destined to show how the programs of the SCOTCH project interoperate and can be combined. It is supposed that the current directory is directory "scotch_7.0" of the SCOTCH distribution. Character "%" represents the shell prompt.

- Partition source graph brol.grf into 7 parts, and save the result to file /tmp/brol.map.

  ```
  % echo cmplt 7 > /tmp/k7.tgt
  % gmap brol.grf /tmp/k7.tgt /tmp/brol.map
  ```

  This can also be done in a single piped command:

  ```
  % echo cmplt 7 | gmap brol.grf - /tmp/brol.map
  ```

  If compressed data handling is enabled, read the graph as a gzip compressed file, and output the mapping as a bzip2 file, on the fly:

  ```
  % echo cmplt 7 | gmap brol.grf.gz - /tmp/brol.map.bz2
  ```

- Partition source graph `brol.grf` into two uneven parts of respective weights $\frac{4}{11}$ and $\frac{7}{11}$, and save the result to file `/tmp/brol.map`.

  ```
  % echo cmpltw 2 4 7 > /tmp/k2w.tgt
  % gmap brol.grf /tmp/k2w.tgt /tmp/brol.map
  ```

  This can also be done in a single piped command:

  ```
  % echo cmpltw 2 4 7 | gmap brol.grf - /tmp/brol.map
  ```

  If compressed data handling is enabled, use `gzip` compressed streams on the fly:

  ```
  % echo cmpltw 2 4 7 | gmap brol.grf.gz -
  /tmp/brol.map.gz
  ```

- Map a 32 by 32 bidimensional grid source graph onto a 256-node hypercube, and save the result to file `/tmp/brol.map`.

  ```
  % gmk_m2 32 32 | gmap - tgt/h8.tgt /tmp/brol.map
  ```

- Build the VTK file `brol.vtk` that contains the display of a source graph the topology and geometry files of which are named `brol.grf` and `brol.xyz`, respectively, to be displayed using a visualization software such as `paraview`.

  ```
  % gout -Mn -Ov brol.grf brol.xyz - /tmp/brol.vtk
  ```

  Although no mapping data is required because of the "`-Mn`" option, note the presence of the dummy input mapping file name "`-`", which is needed to specify the output visualization file name after it.

- Given the topology and geometry files `brol.grf` and `brol.xyz` of a source graph, map the graph on a 8 by 8 bidimensional mesh and display the mapping result on a color screen by means of the public-domain `ghostview` PostScript previewer.

  ```
  % gmap brol.grf tgt/m8x8.tgt | gout brol.grf brol.xyz
  '-Op{c,f,l}' | ghostview -
  ```

- Given the topology and geometry files `brol.grf` and `brol.xyz` of a source graph, partition the graph into 7 parts and create a VTK file to be displayed using a visualization software such as `paraview`.

  ```
  % gpart 7 brol.grf | gout brol.grf brol.xyz '-Ov'
  /tmp/brol_k7.vtk
  ```

- Build a 24-node Cube-Connected-Cycles graph target architecture which will be frequently used. Then, map compressed source file `brol.grf.gz` onto it, and save the result to file `/tmp/brol.map`.

  ```
  % amk_ccc 3 | acpl - /tmp/ccc3.tgt
  % gunzip -c brol.grf.gz | gmap - /tmp/ccc3.tgt
  /tmp/brol.map
  ```

To speed up target architecture loading in the future, the decomposition-defined target architecture is compiled by means of `acpl`.

- Build an architecture graph which is the subgraph of the 8-node de Bruijn graph restricted to vertices labeled 1, 2, 4, 5, 6, map graph `brol.grf` onto it, and save the result to file `/tmp/brol.map`.

```
% (gmk_ub2 3; echo 5 1 2 4 5 6) | amk_grf -L | gmap
brol.grf - /tmp/brol.map
```

Note how the two input streams of program `amk_grf` (that is, the de Bruijn source graph and the five-elements vertex label list) are concatenated into a single stream to be read from the standard input.

- Compile and link the user application `brol.c` with the libScotch library, using the default error handler.

```
% cc brol.c -o brol -lscotch -lscotcherr -lm
```

Note that the mathematical library should also be included, after all of the Scotch libraries.

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

```
% cc brol.c -o brol -I${metisdir} -lscotchmetis
-lscotch -lscotcherr -lmetis -lm
```

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

11 Adding new features to Scotch

Since Scotch is free/libre software, users have the ability to add new features to it. Moreover, as Scotch is intended to be a testbed for new partitioning and ordering algorithms, it has been developed in a very modular way, to ease the development and inclusion of new partitioning and ordering methods to be called within Scotch strategies.

All of the source code for partitioning and ordering methods for graphs and meshes is located in the `src/libscotch/` source subdirectory. Source file names have a very regular pattern, based on the internal data structures they handle.

11.1 Graphs and meshes

The basic structures in Scotch are the `Graph` and `Mesh` structures, which model a simple symmetric graph the definition of which is given in file `graph.h`, and a

simple mesh, in the form of a bipartite graph, the definition of which is given in file `mesh.h`, respectively. From this structure are derived enriched graph and mesh structures:

- `Bgraph`, in file `bgraph.h`: graph with bipartition, that is, edge separation, information attached to it;
- `Kgraph`, in file `kgraph.h`: graph with mapping information attached to it;
- `Hgraph`, in file `hgraph.h`: graph with halo information attached to it, for computing graph orderings;
- `Vgraph`, in file `vgraph.h`: graph with vertex bipartition information attached to it;
- `Hmesh`, in file `hmesh.h`: mesh with halo information attached to it, for computing mesh orderings;
- `Vmesh`, in file `vmesh.h`: graph with vertex bipartition information attached to it.

As version 7.0 of the libScotch does not provide mesh mapping capabilities, neither `Bmesh` nor `Kmesh` structures have been defined to date, but this work is in progress, and these features should be available in the upcoming releases.

All of the structures are in fact defined as `typedef`ed types.

11.2 Methods and partition data

Methods are routines which take one of the above structures as input, and update the fields of the given structure according to the implemented algorithm. Initial methods will behave irrespective of the former values of the structure (like graph growing methods, which compute partitions from scratch), while refinement methods must be provided an existing partition to improve.

In addition to the topological description of the underlying graph, the working graph and mesh structures comprise variables describing the current state of the vertex or edge partition. In all cases is provided a partition array called `parttax`, of size equal to the number of graph vertices, which tells which part every vertex is assigned to. Other variables comprise the communication load and the load imbalance of the current cut, that is, all of the data necessary to measure the quality of a partition. Some other data are also often provided, such as the number of vertices in each part and the list of frontier vertices. They are not relevant to measure the quality of the partition, but to improve the speed of computations. They are used for instance in the multilevel algorithms to compute incremental updates of the current partition state, without having to recompute these values from scratch by considering all of the graph vertices. Implementers of new methods are highly encouraged to use these variables to speed-up their computations, taking examples on typical algorithms such as the multilevel or Fiduccia-Mattheyses ones.

11.3 Adding a new method to Scotch

We will assume in this section that the new method to add is a graph separation method. The procedure explained below is exactly the same for graph bipartitioning, graph mapping, graph ordering, mesh separation, or mesh ordering methods.

Please proceed as explained below.

1. Write the code of the method itself. First, choose a free two-letter code to describe your method, say "xy". In the `libscotch` source directory, create files `vgraph_separate_xy.c` and `vgraph_separate_xy.h`, basing on existing files such as `vgraph_separate_gg.c` and `vgraph_separate_gg.h`, for instance.

   If the method is complex, it can be split across several other files, which will be named `vgraph_separate_xy_firstmodulename.c`, `vgraph_separate_xy_secondmodulename.c`, eventually with matching header files.

   If the method has parameters, create a structure called `VgraphSeparateXyParam`, which contains fields of types that can be handled by the strategy parser, such as the `INT` generic integer type (see below), or `double`, for instance.

   The execution of your method should result in the setting or in the updating of the `Vgraph` structure that is passed to it. See its definition in `vgraph.h` and read several simple graph separation methods, such as `vgraph_separate_zr.c`, to figure out what all of its parameters mean.

   At the end of your method, always call, when the `SCOTCH_DEBUG_VGRAPH2` debug flag is set, the `vgraphCheck` routine, to avoid the spreading of eventual bugs to other parts of the LIBSCOTCH library.

2. Add the method to the parser tables. The files to update are `vgraph_separate_st.c` and `vgraph_separate_st.h`, where "st" stands for "strategy".

   First, edit `vgraph_separate_st.h`. In the `VgraphSeparateStMethodType` enumeration, add a line for your new method `VGRAPHSEPASTMETHXY`. Then, edit `vgraph_separate_st.c`, where all of the remaining actions take place.

   In the top of the file, add a `#include` directive to include `vgraph_separate_xy.h`.

   If the method has parameters, create a `vgraphseparatedefaultxy` C union, basing on an existing one, and fill it with the default values of your method parameters.

   In the `vgraphseparatestmethtab` method array, add a line for the new method. To do so, choose a free single-letter code that will be used to designate the new method in strategy strings. If the method has parameters, the last field should be a pointer to the default structure, else it should be set to `NULL`.

   If the method has parameters, update the `vgraphseparatestparatab` parameter array. Add one data block per parameter. The first field is the name of the method to which the parameter applies, that is, `VGRAPHSEPASTMETHXY`. The second field is the type of the parameter, which can be:

   - `STRATPARAMCASE`: the support type is an `int`. It receives the index in the case string, which is provided as the last field of the parameter line, of the given case character;
   - `STRATPARAMDOUBLE`: the support type is a `double`;
   - `STRATPARAMINT`: the support type is an `INT`, which is the generic integer type handled internally by SCOTCH. This type has variable extent, depending on compilation flags, as described in Section 8.1.5;

- STRATPARAMSTRING: a (small) character string;
- STRATPARAMSTRAT: strategy. For instance, the graph ordering method by nested dissection takes a vertex partitioning strategy as one of its parameters, to compute the vertex separators.

The fourth and fifth fields are the address of the location of the default structure and the address of the parameter within this default structure, respectively. From these two values can be computed at run time the offset of the parameter within any instance of the parameter structure, which is used to fill the actual structures in the parsed strategy evaluation tree. The value of the sixth parameter depends on the type of the parameter. It should be NULL for STRATPARAMDOUBLE and STRATPARAMINT parameters, points to the string of available case letters for STRATPARAMCASE parameters, points to the target string buffer for STRATPARAMSTRING parameters, and points to the relevant method parsing table for STRATPARAMSTRAT parameters.

3. Edit the makefile of the libScotch source directory to enable the compilation and linking of the method. Depending on libScotch versions, this makefile is either called Makefile or make_gen.

4. Compile in debug mode and experiment with your routine, by creating strategies that contain its single-letter code.

5. To change the default strategy string used by the libScotch library, update file library_graph_order.c, since it is the graph ordering routine which makes use of graph vertex separation methods to compute separators for the nested dissection ordering method.

11.4 Licensing of new methods and of derived works

According to the terms of the CeCILL-C license [7] under which the Scotch software package is distributed, the works that are carried out to improve and extend the libScotch library must be licensed under the same terms. Basically, it means that you will have to distribute the sources of your new methods, along with the sources of Scotch, to any recipient of your modified version of the libScotch, and that you grant these recipients the same rights of update and redistribution as the ones that are given to you under the terms of CeCILL-C. Please read it carefully to know what you can do and cannot do with the Scotch distribution.

You should have received a copy of the CeCILL-C license along with the Scotch distribution; if not, please browse the CeCILL website at http://www.cecill.info/licenses.en.html.

Credits

I wish to thank all of the following people:

- Patrick Amestoy collaborated to the design of the Halo Approximate Minimum Degree algorithm [49] that had been embedded into Scotch 3.3, and provided versions of his Approximate Minimum Degree algorithm, available

since version 3.2, and of his Halo Approximate Minimum Fill algorithm, available since version 3.4. He designed the mesh versions of the approximate minimum degree and approximate minimum fill algorithms, which are available since version 4.0;

- Clément Barthélemy improved the Windows compilation system and coded the Windows threading module, available since version 7.0.5. As the Scotch Consortium core engineer, he also contributed to various improvements;
- Sébastien Fourestier coded the mapping with fixed vertices, remapping, and remapping with fixed vertices sequential routines that are available since version 6.0;
- Marc Fuentes designed the mesh-to-dual-graph routines, extended the functional scope of the MeTiS compatibility library, and coded CMake environment files;
- Jun-Ho Her coded the graph partitioning with overlap routines that were introduced in the unpublished 5.2 release, subsequently publicly released in version 6.0;
- Amaury Jacques improved the development environment and contributed to the consistency checking and non-regression testing routines;
- Cédric Lachat contributed to the robustness of the software, in relation with his development of the PaMPA software;
- Connor Alexander Mayon coded the genetic algorithm partitioning methods, available since version 7.0.10;
- Xavier Muller contributed to the compilation chain;
- Tetsuya Mishima contributed to the Windows threading module, available since version 7.0.5;
- Charles Pellegrini contributed to bug hunting and software quality in branch 7.0;
- Alex Pothen kindly provided a version of his Multiple Minimum Degree algorithm, which was embedded into Scotch from version 3.2 to version 3.4;
- Florent Pruvost set up the continuous integration environment on Inria GitLab;
- Luca Scarano coded the multilevel graph algorithm in Scotch 3.1;
- Yves Secretan contributed to the MinGW32 port;
- David Sherman proofread version 3.2 of this manual;
- Dimitri Walther provided insightful comments, feedback and patches for the static mapping module.

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. Compressed graphs and the minimum degree algorithm. *SIAM J. Sci. Comput.*, 16(6):1404–1411, 1995.

[3] 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.

[4] D. Aubert et al. Tulip. Available from `https://tulip.labri.fr/`.

[5] 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.

[6] R. F. Boisvert, R. Pozo, and K. A. Remington. The Matrix Market exchange formats: initial design. NISTIR 5935, National Institute of Standards and Technology, December 1996.

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

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

[9] 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.

[10] I. Duff. On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Software*, 7(3):315–330, September 1981.

[11] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection. Technical Report TR/PA/92/86, CERFACS, Toulouse, France, October 1992.

[12] 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.

[13] 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.

[14] S. Fourestier and F. Pellegrini. Adaptation au repartitionnement de graphes d'une méthode d'optimisation globale par diffusion. In *Proc. RenPar'20, Saint-Malo, France*, May 2011.

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

[16] 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.

[17] 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.

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

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

[20] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. *ACM Trans. Math. Software*, 2:322–330, 1976.

[21] 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.

[22] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Trans. Parallel Distrib. Syst.*, 8(5):502–520, 1997.

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

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

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

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

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

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

[29] 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.

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

[31] P. Hénon, F. Pellegrini, P. Ramet, J. Roman, and Y. Saad. High performance complete and incomplete factorizations for very large sparse systems by using SCOTCH and PASTIX softwares. In *Proc. 11^{th} SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, USA*, February 2004.

[32] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2(4):225–231, December 1973.

[33] 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.

[34] G. Karypis and V. Kumar. MeTiS – unstructured graph partitioning and sparse matrix ordering system – version 2.0. Technical report, University of Minnesota, June 1995.

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

[36] 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.

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

[38] Kitware. Paraview. Available from `https://www.paraview.org/`.

[39] M. Laguna, T. A. Feo, and H. C. Elrod. A greedy randomized adaptative search procedure for the two-partition problem. *Operations Research*, pages 677–687, July 1994.

[40] C. Leiserson and J. Lewis. Orderings for parallel sparse symmetric factorization. In *Third SIAM Conference on Parallel Processing for Scientific Computing*, 1987.

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

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

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

[44] 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.

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

[46] 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`.

[47] 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.

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

[49] F. Pellegrini, J. Roman, and P. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. In *Proc. Irregular'99, San Juan*, LNCS 1586, pages 986–995, April 1999.

[50] François Pellegrini and Cédric Lachat. Process Mapping onto Complex Architectures and Partitions Thereof. Research Report RR-9135, Inria Bordeaux Sud-Ouest, December 2017. Available from `https://hal.inria.fr/hal-01671156`.

[51] A. Pothen and C.-J. Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Software*, 16(4):303–324, December 1990.

[52] 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.

[53] E. Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. In *Proc. SHPCC'94, Knoxville*, pages 324–333. IEEE, May 1994.

[54] E. Rothberg and A. Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. In *Supercomputing'93 Proceedings*. IEEE, 1993.

[55] E. Rothberg and R. Schreiber. Improved load distribution in parallel sparse Cholesky factorization. In *Supercomputing'94 Proceedings*. IEEE, 1994.

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

[57] Inc. Silicon Graphics. Open Inventor. Available from `http://oss.sgi.com/projects/inventor/`.

[58] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2:135–148, 1991.

[59] 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.

[60] 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*, number 980 in LNCS, pages 121–126, 1995.