Scotch and libScotch 5.1 User’s Guide

(version 5.1.11)

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

November 17, 2010

Abstract

This document describes the capabilities and operations of Scotch and libScotch, a software package and a software library devoted to static mapping, 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 6
  1.1 Static mapping ............................................. 6
  1.2 Sparse matrix ordering .................................... 6
  1.3 Contents of this document ................................ 7

2 The Scotch project 7
  2.1 Description .................................................. 7
  2.2 Availability ................................................ 8

3 Algorithms 8
  3.1 Static mapping by Dual Recursive Bipartitioning .............. 8
    3.1.1 Static mapping ........................................... 8
    3.1.2 Cost function and performance criteria .................... 9
    3.1.3 The Dual Recursive Bipartitioning algorithm ............... 10
    3.1.4 Partial cost function .................................... 11
    3.1.5 Execution scheme ........................................ 11
    3.1.6 Graph bipartitioning methods ............................. 12
    3.1.7 Mapping onto variable-sized architectures ................. 15
  3.2 Sparse matrix ordering by hybrid incomplete nested dissection 15
    3.2.1 Minimum Degree ......................................... 15
    3.2.2 Nested dissection ........................................ 15
    3.2.3 Hybridization .......................................... 16
    3.2.4 Performance criteria .................................... 16
    3.2.5 Ordering methods ....................................... 17
    3.2.6 Graph separation methods ............................... 18

4 Updates 18
  4.1 Changes from version 4.0 .................................. 18
  4.2 Changes from version 5.0 .................................. 19

5 Files and data structures 19
  5.1 Graph files .................................................. 19
  5.2 Mesh files .................................................. 20
  5.3 Geometry files ............................................. 22
  5.4 Target files ................................................ 22
    5.4.1 Decomposition-defined architecture files .................. 23
    5.4.2 Algorithmically-coded architecture files .................. 23
    5.4.3 Variable-sized architecture files ........................ 25
  5.5 Mapping files ............................................... 26
  5.6 Ordering files ............................................. 26
  5.7 Vertex list files .......................................... 27

6 Programs 27
  6.1 Invocation .................................................. 28
  6.2 Using compressed files ...................................... 28
  6.3 Description ................................................ 30
    6.3.1 acpl ..................................................... 30
    6.3.2 amk_# .................................................... 30
    6.3.3 amk_grf ............................................... 32
    6.3.4 atst ..................................................... 33
6.3.5 gcv ................................................. 33
6.3.6 gmap / gpart ...................................... 34
6.3.7 gmkg* .............................................. 36
6.3.8 gmk_msh ........................................... 37
6.3.9 gmtst ................................................ 37
6.3.10 gord ............................................... 38
6.3.11 gotst ............................................... 40
6.3.12 gout ............................................... 40
6.3.13 gtst ............................................... 43
6.3.14 mcv ............................................... 43
6.3.15 mmk* .............................................. 44
6.3.16 mord ............................................... 45
6.3.17 mtst ............................................... 46

7 Library .............................................. 47
  7.1 Calling the routines of LIBSCOTCH ................. 47
    7.1.1 Calling from C .................................. 47
    7.1.2 Calling from Fortran ............................. 48
    7.1.3 Compiling and linking ............................ 49
    7.1.4 Dynamic library issues ........................... 49
    7.1.5 Machine word size issues ....................... 49
  7.2 Data formats .................................... 50
    7.2.1 Architecture format ............................. 50
    7.2.2 Graph format ................................... 50
    7.2.3 Mesh format .................................... 53
    7.2.4 Geometry format ................................ 54
    7.2.5 Block ordering format ........................... 56
  7.3 Strategy strings ................................ 57
    7.3.1 Using default strategy strings ................. 57
    7.3.2 Mapping strategy strings ....................... 58
    7.3.3 Graph bipartitioning strategy strings ......... 59
    7.3.4 Ordering strategy strings ..................... 63
    7.3.5 Node separation strategy strings ............... 66
  7.4 Target architecture handling routines .......... 70
    7.4.1 SCOTCH_archInit ................................. 70
    7.4.2 SCOTCH_archExit ................................ 70
    7.4.3 SCOTCH_archLoad ................................ 71
    7.4.4 SCOTCH_archSave ................................ 71
    7.4.5 SCOTCH_archBuild ................................ 72
    7.4.6 SCOTCH_archCmplt ................................ 72
    7.4.7 SCOTCH_archCmpltw .............................. 73
    7.4.8 SCOTCH_archName ................................ 73
    7.4.9 SCOTCH_archSize ................................ 74
    7.4.10 SCOTCH_archHcub ................................ 74
    7.4.11 SCOTCH_archMesh2D ................................ 75
    7.4.12 SCOTCH_archMesh3D ................................ 75
    7.4.13 SCOTCH_archTleaf ................................ 76
    7.4.14 SCOTCH_archTorus2D ............................... 76
    7.4.15 SCOTCH_archTorus3D ............................... 77
  7.5 Graph handling routines .......................... 77
    7.5.1 SCOTCH_graphAlloc ............................... 77
7.5.2 SCOTCH_graphInit ............................................. 77
7.5.3 SCOTCH_graphExit ............................................. 78
7.5.4 SCOTCH_graphFree ............................................. 78
7.5.5 SCOTCH_graphLoad ............................................. 79
7.5.6 SCOTCH_graphSave ............................................. 79
7.5.7 SCOTCH_graphBuild ............................................ 80
7.5.8 SCOTCH_graphBase ............................................. 81
7.5.9 SCOTCH_graphCheck ............................................. 82
7.5.10 SCOTCH_graphSize ............................................ 82
7.5.11 SCOTCH_graphData ............................................. 83
7.5.12 SCOTCH_graphStat ............................................. 84

7.6 Graph mapping and partitioning routines ................................ 85
7.6.1 SCOTCH_graphPart .............................................. 85
7.6.2 SCOTCH_graphMap .............................................. 86
7.6.3 SCOTCH_graphMapInit ........................................... 87
7.6.4 SCOTCH_graphMapExit .......................................... 87
7.6.5 SCOTCH_graphMapLoad ......................................... 88
7.6.6 SCOTCH_graphMapSave ......................................... 88
7.6.7 SCOTCH_graphMapCompute ...................................... 89
7.6.8 SCOTCH_graphMapView .......................................... 89

7.7 Graph ordering routines ........................................ 90
7.7.1 SCOTCH_graphOrder ............................................ 90
7.7.2 SCOTCH_graphOrderInit ....................................... 91
7.7.3 SCOTCH_graphOrderExit ....................................... 92
7.7.4 SCOTCH_graphOrderLoad ...................................... 92
7.7.5 SCOTCH_graphOrderSave ...................................... 93
7.7.6 SCOTCH_graphOrderSaveMap .................................. 93
7.7.7 SCOTCH_graphOrderSaveTree .................................. 94
7.7.8 SCOTCH_graphOrderCheck ..................................... 94
7.7.9 SCOTCH_graphOrderCompute ................................... 95
7.7.10 SCOTCH_graphOrderComputeList ................................ 95

7.8 Mesh handling routines .......................................... 96
7.8.1 SCOTCH_meshAlloc ............................................ 96
7.8.2 SCOTCH_meshInit .............................................. 97
7.8.3 SCOTCH_meshExit .............................................. 97
7.8.4 SCOTCH_meshLoad .............................................. 97
7.8.5 SCOTCH_meshSave .............................................. 98
7.8.6 SCOTCH_meshBuild ............................................ 98
7.8.7 SCOTCH_meshCheck ............................................ 100
7.8.8 SCOTCH_meshSize ............................................. 100
7.8.9 SCOTCH_meshData ............................................. 101
7.8.10 SCOTCH_meshStat ............................................ 102
7.8.11 SCOTCH_meshGraph ........................................... 103

7.9 Mesh ordering routines .......................................... 104
7.9.1 SCOTCH_meshOrder ............................................ 104
7.9.2 SCOTCH_meshOrderInit ....................................... 105
7.9.3 SCOTCH_meshOrderExit ....................................... 106
7.9.4 SCOTCH_meshOrderSave ...................................... 107
7.9.5 SCOTCH_meshOrderSaveMap .................................. 107
7.9.6 SCOTCH_meshOrderSaveTree .................................. 108
7.9.7 SCOTCH_meshOrderCheck ..................................... 108
7.9.8 SCOTCH_meshOrderCompute .................................. 109
7.10 Strategy handling routines .................................... 109
  7.10.1 SCOTCH_stratAlloc ................................... 109
  7.10.2 SCOTCH_stratInit ................................... 110
  7.10.3 SCOTCH_stratExit ................................ 110
  7.10.4 SCOTCH_stratSave ................................ 110
  7.10.5 SCOTCH_stratGraphBipart .............................. 111
  7.10.6 SCOTCH_stratGraphMap ................................ 111
  7.10.7 SCOTCH_stratGraphMapBuild ............................ 112
  7.10.8 SCOTCH_stratGraphOrder ............................... 113
  7.10.9 SCOTCH_stratGraphOrderBuild .......................... 113
  7.10.10 SCOTCH_stratMeshOrder ............................... 114
  7.10.11 SCOTCH_stratMeshOrderBuild .......................... 114
7.11 Geometry handling routines .................................. 115
  7.11.1 SCOTCH_geomAlloc ................................... 115
  7.11.2 SCOTCH_geomInit ................................... 115
  7.11.3 SCOTCH_geomExit .................................. 116
  7.11.4 SCOTCH_geomData ................................... 116
  7.11.5 SCOTCH_graphGeomLoadChac ........................... 117
  7.11.6 SCOTCH_graphGeomSaveChac ........................... 117
  7.11.7 SCOTCH_graphGeomLoadHabo ........................... 118
  7.11.8 SCOTCH_graphGeomLoadScot ........................... 119
  7.11.9 SCOTCH_graphGeomSaveScot ........................... 119
  7.11.10 SCOTCH_meshGeomLoadHabo ........................... 120
  7.11.11 SCOTCH_meshGeomLoadScot ........................... 121
  7.11.12 SCOTCH_meshGeomSaveScot ........................... 121
7.12 Other data structure routines ................................ 122
  7.12.1 SCOTCH_mapAlloc ................................... 122
  7.12.2 SCOTCH_orderAlloc ................................ 122
7.13 Error handling routines ..................................... 123
  7.13.1 SCOTCH_errorPrint .................................. 123
  7.13.2 SCOTCH_errorPrintW ................................ 123
  7.13.3 SCOTCH_errorProg .................................. 123
7.14 Miscellaneous routines ....................................... 124
  7.14.1 SCOTCH_randomReset .................................. 124
  7.14.2 SCOTCH_version ...................................... 124
7.15 MexIS compatibility library .................................. 124
  7.15.1 METIS_EdgeND ...................................... 125
  7.15.2 METIS_NodeND ...................................... 125
  7.15.3 METIS_NodeWND ...................................... 126
  7.15.4 METIS_PartGraphKway ................................ 127
  7.15.5 METIS_PartGraphRecursive ............................ 128
  7.15.6 METIS_PartGraphVKway ................................ 128

8 Installation ......................................................... 129
  8.1 Thread issues ............................................. 130
  8.2 File compression issues ................................... 130
  8.3 Machine word size issues ................................... 130

9 Examples .......................................................... 131
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 [13]. 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 [11, 21]. 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 [4, 22, 31, 33, 49]. 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 [21, 56].

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

2 The SCOTCH project

2.1 Description

SCOTCH is a project carried out at the Laboratoire Bordelais de Recherche en Informatique (LaBRI) of the Université Bordeaux I, and now within the Bacchus project of INRIA Bordeaux Sud-Ouest. Its goal is to study the applications of graph theory to scientific computing, using a “divide and conquer” approach.

It focused first on static mapping, and has resulted in the development of the Dual Recursive Bipartitioning (or DRB) mapping algorithm and in the study of several graph bipartitioning heuristics [41], all of which have been implemented in the SCOTCH software package [45]. 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 [46, 47]. More recently, the ordering capabilities of SCOTCH have been extended to native mesh structures, thanks to hypergraph partitioning algorithms. New graph partitioning methods have also been recently added [8, 42].

Version 5.0 of SCOTCH is 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 5.1 User’s Guide [43].
2.2 Availability
Starting from version 4.0, which has been developed at INRIA within the ScAlAp-plix project, SCOTCH is available under a dual licensing basis. On the one hand, it is downloadable from the SCOTCH web page as free/libre software, to all interested parties willing to use it as a library or to contribute to it as a testbed for new partitioning and ordering methods. On the other hand, it can also be distributed, under other types of licenses and conditions, to parties willing to embed it tightly into closed, proprietary software.

The free/libre software license under which SCOTCH 5.1 is distributed is the CeCILL-C license [6], 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 8 to see how to obtain the free/libre distribution of SCOTCH.

3 Algorithms

3.1 Static mapping by Dual Recursive Bipartitioning
For a detailed description of the mapping algorithm and an extensive analysis of its performance, please refer to [41, 44]. In the next sections, we will only outline the most important aspects of the algorithm.

3.1.1 Static mapping
The parallel program to be mapped onto the target architecture is modeled by a valued 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 valued 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 $\tau_{S,T} : V(S) \rightarrow V(T)$ and $\rho_{S,T} : E(S) \rightarrow P(E(T))$, where $P(E(T))$ 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 $\rho_{S,T}(e_S) = \{e^1_T, e^2_T, \ldots, e^n_T\}$ if communication channel $e_S$ of $S$ is routed through communication links $e^1_T$, $e^2_T$, $\ldots$, $e^n_T$ of $T$. $|\rho_{S,T}(e_S)|$ denotes the dilation of edge $e_S$, that is, the number of edges of $E(T)$ used to route $e_S$. 
3.1.2 Cost function and performance criteria

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

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

This function, which has already been considered by several authors for hypercube target topologies [11, 21, 25], has several interesting properties: it is easy to compute, allows incremental updates performed by iterative algorithms, and its minimization favors the mapping of intensively intercommunicating processes onto nearby processors; regardless of the type of routage implemented on the target machine (store-and-forward or cut-through), it models the traffic on the interconnection network and thus the risk of congestion.

The strong positive correlation between values of this function and effective execution times has been experimentally verified by Hammond [21] on the CM-2, and by Hendrickson and Leland [26] on the nCUBE 2.

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

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

$$\mu_{\text{map}} \overset{\text{def}}{=} \frac{\sum_{v \in V(S)} w_S(v)}{\sum_{v \in V(T)} w_T(v)}$$

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

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 $\mu_{\text{exp}}$, the average edge expansion, which can be compared to $\mu_{\text{dil}}$, the average edge dilation; these are defined as

$$\mu_{\text{exp}} \overset{\text{def}}{=} \frac{f_C(e)}{w_S(e_S)}$$

$$\mu_{\text{dil}} \overset{\text{def}}{=} \frac{\sum_{e \in E(S)} |\rho_{S,T}(e_S)|}{|E(S)|} .$$
\( \delta_{\text{exp}} \triangleq \frac{H_{\text{exp}}}{H_{\text{dil}}} \) is smaller than 1 when the mapper succeeds in putting heavily inter-
communicating processes closer to each other than it does for lightly communicating
processes; they are equal if all edges have same weight.

3.1.3 The Dual Recursive Bipartitioning algorithm

Our mapping algorithm uses a *divide and conquer* approach to recursively allocate
subsets of processes to subsets of processors [41]. 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.

```c
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 archi-
tecture;

- **a domain bipartitioning function**, which, given a domain, bipartitions it in two
disjoint subdomains;

- **a domain distance function**, which gives, in the target graph, a measure of the
distance between two disjoint domains. Since domains may not be convex nor
connected, this distance may be estimated. However, it must respect certain
homogeneity properties, such as giving more accurate results as domain sizes
decrease. The domain distance function is used by the graph bipartitioning
algorithms to compute the communication function to minimize, since it allows
the mapper to estimate the dilation of the edges that link vertices which belong
to different domains. Using such a distance function amounts to considering
that all routings will use shortest paths on the target architecture, which
is how most parallel machines actually do. We have thus chosen that our
program would not provide routings for the communication channels, leaving
their handling to the communication system of the target machine;

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

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

3.1.4 Partial cost function

The production of efficient complete mappings requires that all graph bipartition-
ings 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

$$f'_C(\tau_{S,T}, \rho_{S,T}) \overset{\text{def}}{=} \sum_{v \in V(S')} \left\{ v, v' \right\} \in E(S') \left| \rho_{S,T}(\left\{ v, v' \right\}) \right|,$$

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 pos-
sible to avoid local choices that might prove globally bad, as explained below. This
amounts to incorporating additional constraints to the standard graph bipartition-
ing problem, turning it into a more general optimization problem termed skewed
graph partitioning by some authors [27].

![Figure 1](image)

**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.

3.1.5 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 [44].

![Figure 2](image_url)

Figure 2: Influence of depth-first and breadth-first sequenceings 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”).

3.1.6 Graph bipartitioning methods

The core of our recursive 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 mapping strategies. The currently implemented graph bipartitioning methods are listed below.

**Band**

Like the multi-level method which will be described below, the band method is a meta-algorithm, in the sense that it does not itself compute partitions, but rather helps other partitioning algorithms perform better. It is a refinement
algorithm which, from a given initial partition, extracts a band graph of given width (which only contains graph vertices that are at most at this distance from the separator), calls a partitioning strategy on this band graph, and 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 multi-level context, it also improves partition quality, by coercing partitions in a problem space that derives from the one which was globally defined at the coarsest level, thus preventing local optimization refinement algorithms to be trapped in local optima of the finer graphs [8].

Diffusion
This global optimization method, presented in [42], 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 [12] is an almost-linear improvement of the famous Kernighan-Lin algorithm [35]. 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 [44].

As several authors quoted before [24, 32], the Fiduccia-Mattheyses algorithm gives better results when trying to optimize a good starting partition. There-
fore, 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 [18]. 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 [17], and by Simon in [54].

**Greedy graph growing**

This greedy algorithm, which has been proposed by Karypis and Kumar [31], belongs to the GRASP ("Greedy Randomized Adaptive Search Procedure") class [36]. 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.

**Multi-level**

This algorithm, which has been studied by several authors [4, 23, 31] and should be considered as a strategy rather than as a method since it uses other methods as parameters, repeatedly reduces the size of the graph to bipartition by finding matchings that collapse vertices and edges, computes a partition for the coarsest graph obtained, and projects the result back to the original graph, as shown in Figure 3. The multi-level method, when used in conjunc-

![Multi-level Partitioning Process](image)

Figure 3: The multi-level partitioning process. In the uncoarsening phase, the light and bold lines represent for each level the projected partition obtained from the coarser graph, and the partition obtained after refinement, respectively.
tion with the Fiduccia-Mattheyses method to compute the initial partitions and refine the projected partitions at every level, usually leads to a significant improvement in quality with respect to the plain Fiduccia-Mattheyses method. By coarsening the graph used by the Fiduccia-Mattheyses method to compute and project back the initial partition, the multi-level algorithm broadens the scope of the Fiduccia-Mattheyses algorithm, and makes possible for it to account for topological structures of the graph that would else be of a too high level for it to encompass in its local optimization process.

3.1.7 Mapping onto variable-sized architectures

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

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

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

3.2 Sparse matrix ordering by hybrid incomplete nested dissection

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

3.2.1 Minimum Degree

The minimum degree algorithm [55] 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, 16, 39]). However, the algorithm is intrinsically sequential, and very little can be theoretically proved about its efficiency.

3.2.2 Nested dissection

The nested dissection algorithm [17] 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 $B$.

Many theoretical results have been carried out on nested dissection ordering [7, 38], 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 [28, 37], or from edge separators [48, and included references] by minimum cover techniques [9, 30], but other techniques such as spectral vertex partitioning have also been used [49].

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 [19, 31, 46] to the ones obtained with the minimum degree algorithm [39]. 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, 14, 15, 19, 46, 53, and included references].

3.2.3 Hybridization

Due to their complementary nature, several schemes have been proposed to hybridize the two methods [28, 34, 46]. 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 [20, 50, 51, 52] block solving, to select ordering algorithms that minimize the number of extra-diagonal blocks [7], thus allowing for efficient use of BLAS3 primitives, and to reduce inter-processor communication.

3.2.4 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: \( h_{\text{min}}, h_{\text{max}}, \) and \( h_{\text{avg}} \) denote the minimum, maximum, and average heights of the tree\(^1\), respectively, and \( h_{\text{dlt}} \) is the variance, expressed as a percentage of \( h_{\text{avg}} \). Since small separators result in small chains in the elimination tree, \( h_{\text{avg}} \) should also indirectly reflect the quality of separators.

### 3.2.5 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 \([47]\) 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

---

\(^1\) We do not consider as leaves the disconnected vertices that are present in some meshes, since they do not participate in the solving process.
subgraphs, and specific ordering methods are applied to the separators and to the resulting subgraphs (see sections 3.2.2 and 3.2.3).

3.2.6 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 13.

Greedy graph growing
This is a vertex-oriented version of the edge-oriented greedy graph growing algorithm described in page 14.

Multi-level
This is a vertex-oriented version of the edge-oriented multi-level 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 multi-level 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 [48]. This method requires the computation of maximal matchings in the bipartite graphs associated with the edge cuts, which are built using Duff’s variant [9] of the Hopcroft and Karp algorithm [30]. Edge separators are computed by using a bipartitioning strategy, which can use any of the graph bipartitioning methods described in section 3.1.6, page 12.

4 Updates
4.1 Changes from version 4.0
SCOTCH has gone parallel with the release of PT-SCOTCH, the Parallel Threaded SCOTCH. People interested in these parallel routines should refer to the PT-SCOTCH and libSCOTCH 5.1 User’s Guide [43], which extends this manual.
A compatibility library has been developed to allow users to try and use Scotch in programs that were designed to use MeBIS. Please refer to Section 7.15 for more information.

Scotch can now handle compressed streams on the fly, in several widely used formats such as gzip, bzip2 or lzma. Please refer to Section 6.2 for more information.

### 4.2 Changes 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 8.3 for more information.

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

### 5 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 explicitly stated, **numberings always start from zero**, not one.

#### 5.1 Graph files

Graph files, which usually end in “.grf” or “.src”, describe valued graphs, which can be valued 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 compactity, 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 6.3.5 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 16, page 52, 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 [24], 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.

```
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.

### 5.2 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.

Meshees 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 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 5.1, 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 by 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 [24], 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.
5.3 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.

5.4 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.

5.4.1 Decomposition-defined architecture files

Decomposition-defined architecture files are the standard way to describe weighted and/or irregular target architectures. Several file formats exist, but we only present here the most humanly readable one, which begins in “deco 0” (“deco” stands for “decomposition-defined” architecture, and “0” is the format type).

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.

5.4.2 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.
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.

**cmplt size**
Defines a complete graph with size vertices. The vertex labels are numbers between 0 and size - 1.

**cmpltw size load_0 load_1 ... load_{size-1}**
Defines a weighted complete graph with size vertices. The vertex labels are numbers between 0 and size - 1, and vertices are assigned integer weights in the order in which these are provided.

**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.

**tleaf levnbr sizeval_0 linkval_0 ... sizeval_{levnbr-1} linkval_{levnbr-1}**
Defines a hierarchical, tree-shaped, architecture with levnbr levels and \( \sum_{i=0}^{levnbr-1} \) sizeval_0, leaf vertices. This topology is used to model multi-stage, 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 shown in Figure 8. The communication cost between two nodes is the cost of the highest common ancestor level.

The two additional parameters cluster and weight serve to model heterogeneous architectures for which multiprocessor nodes having several highly interconnected processors (typically by means of shared memory) are linked by means of networks of lower bandwidth. cluster represents the number of levels to traverse, starting from the root of the leaf, before reaching the multiprocessors, each multiprocessor having \( \text{height-}\text{cluster} \) nodes. weight is the relative cost of extra-cluster links, that is, links in the upper levels of the tree-leaf graph. Links within clusters are assumed to have weight 1.
When there are no clusters at all, that is, in the case of purely homogeneous architectures, set cluster to be equal to height, and weight to 1.

\textbf{mesh2D} $dimX \ dimY$

Defines a bidimensional array of $dimX$ columns by $dimY$ rows. The vertex with coordinates $(posX, posY)$ has label $posY \times dimX + posX$.

\textbf{mesh3D} $dimX \ dimY \ dimZ$

Defines a tridimensional array of $dimX$ columns by $dimY$ rows by $dimZ$ levels. The vertex with coordinates $(posX, posY, posZ)$ has label $(posZ \times dimY + posY) \times dimX + posX$.

\textbf{torus2D} $dimX \ dimY$

Defines a bidimensional array of $dimX$ columns by $dimY$ rows, with wraparound edges. The vertex with coordinates $(posX, posY)$ has label $posY \times dimX + posX$.

\textbf{torus3D} $dimX \ dimY \ dimZ$

Defines a tridimensional array of $dimX$ columns by $dimY$ rows by $dimZ$ levels, with wraparound edges. The vertex with coordinates $(posX, posY, posZ)$ has label $(posZ \times dimY + posY) \times dimX + posX$.

\section{5.4.3 Variable-sized architecture files}

Variable-sized architectures are a class of algorithmically-coded architectures the size of which is not defined \textit{a priori}. 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.

\textbf{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.

\textbf{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 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.

5.5 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 9 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 9: 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.

5.6 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 10 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 10: 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.

5.7 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 11 shows the list made from three vertices of labels 2, 45, and 7.

```
3  2  45  7
```

Figure 11: Example of vertex list with three vertices of labels 2, 45, and 7.

6 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 12.

6.1 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. 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 9.

Error messages are standardized, but may not be fully explanatory. However, most of the errors you may run into should be related to file formats, and located in “...Load” routines. In this case, compare your data formats with the definitions given in section 5, and use the gttst and mtst programs to check the consistency of source graphs and meshes.

6.2 Using compressed files

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

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

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

When the compressed format allows it, several files can be provided on the same stream, and be uncompressed on the fly. For instance, the command “cat brol.grf.gz brol.xyz.gz | gout -.gz -.gz -Mn - brol.iv” concatenates the topology and geometry data of some graph brol and feed them
Figure 12: General architecture of the SCOTCH project. All of the features offered by the stand-alone programs are also available in the LIBSCOTCH library.
as a single compressed stream to the standard input of program gout, hence the
"-.gz" to indicate a compressed standard stream.

6.3 Description
6.3.1 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 5.4.1, page 23, 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.

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

Program \texttt{amk\_fft2} outputs the target architecture file of a binary Fast-
Fourier-Transform graph of dimension \textit{dim}. Vertex \((l, m)\) of FFT(\textit{dim}),
with \(0 \leq l \leq \textit{dim}\) and \(0 \leq m < 2^{\textit{dim}}\), is linked to vertices \((l - 1, m),
(l - 1, m \mod 2^{l-1}), (l + 1, m)\), and \((l + 1, m \oplus 2^l)\), if they exist, and is labeled
\(l \times 2^{\textit{dim}} + m\).

Program \texttt{amk\_hy} outputs the target architecture file of a hypercube graph
of dimension \textit{dim}. Vertices are labeled according to the decimal value of
their binary representation. The decomposition-defined target architectures
computed by \texttt{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 bipartitioning are identical. To achieve best
performance and save space, use the built-in architecture.

Program \texttt{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 13).

Options

\texttt{-h} Display the program synopsis.

\texttt{-m method}
Select the bipartitioning method (for \texttt{amk\_m2} only).

\texttt{n} Nested dissection.

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

\texttt{-V} Print the program version and copyright.
6.3.3 \texttt{amk\_grf}

**Synopsis**

\texttt{amk\_grf [input\_graph\_file [output\_target\_file]] options}

**Description**

The program \texttt{amk\_grf} turns a source graph file into a decomposition-defined target file. It computes a recursive bipartitioning of the source graph, as well as the array of distances between all pairs of its vertices, both of which are combined to give a decomposition-defined target architecture of same topology as the input source graph.

The \texttt{-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 \texttt{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 \texttt{-b} option selects the recursive bipartitioning strategy used to build the decomposition of the source graph. For regular, unweighted, topologies, the \texttt{'-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_archBuild, page 72, for further information.

Options

- `strategy`
  Use recursive bipartitioning strategy `strategy` to build the decomposition of the architecture graph. The format of bipartitioning strategies is defined within section 7.3.2, at page 59.

- `-h`
  Display the program synopsis.

- `-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.

6.3.4 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.

6.3.5 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 [5], the Harwell-Boeing collection format [10], the Chaco/Metis graph format [24], 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 “-Ibo -Os”.

6.3.6 gmap / gpart

Synopsis

\texttt{gmap} [input\_graph\_file [input\_target\_file [output\_mapping\_file [output\_log\_file]]]] options

\texttt{gpart} number\_of\_parts [input\_graph\_file [output\_mapping\_file [output\_log\_file]]]

options

Description

The program \texttt{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 \texttt{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.

The only mapping method implemented in version 5.1 is the Dual Recursive Bipartitioning algorithm, which uses graph bipartitioning methods. Available bipartitioning methods include a multi-level algorithm 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 greedy method derived from the Gibbs, Poole, and
Stockmeyer algorithm, the greedy graph growing heuristic, and a greedy “ex-
actifying” refinement algorithm designed to balance vertex loads as much as
possible; random and backtracking methods are also provided.

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.

If mapping statistics are wanted rather than the mapping output itself, map-
ing output can be set to /dev/null, with option -vmt to get mapping statis-
tics 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 the default mapping strategy according to the given preference
flags. Some of these flags are antagonistic, while others can be combined.
See Section 7.3.1 for more information. The currently available flags are
the following.

b Enforce load balance as much as possible.
q Privilege quality over speed. This is the default behavior.
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 incompat-
ible 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. The format of mapping strategies is de-
defined in section 7.3.2. This option is incompatible with options -b and
-c.

-s obj
Mask source edge and vertex weights. This option allows the user to “un-
weight” weighted source graphs by removing weights from edges and ver-
tices 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 information.
  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.

6.3.7 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 9, 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 mesh with dimX columns and dimY rows. If the -t option is set, tori are built instead of meshes. The vertex of coordinates \((posX, posY)\) is labeled \(posY \times dimX + posX\).

Program gmk_m3 outputs the source file of a tridimensional mesh with dimZ layers of dimY rows by dimX columns. If the -t option is set, tori are built instead of meshes. 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.

36
Options

-g output_geometry_file
Output graph geometry to file output_geometry_file (for gmk_m2 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 mesh (for gmk_m2 only).
-V Print the program version and copyright.

6.3.8 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.

6.3.9 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 commu-
nication.

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 dilatation 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 inputting 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.

6.3.10 `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 [17], but classical [39] and state-of-the-art [1, 47] 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 the default ordering strategy according to the given preference flags.
Some of these flags are antagonistic, while others can be combined. See
Section 7.3.1 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 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 5.5.
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 7.3.4.

-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.
6.3.11 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 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` Print the program version and copyright.

6.3.12 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 as 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** files [40] 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 14.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 14.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 14: PostScript display 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 \(-\text{om}\) 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 \(-\text{om}\) and \(-\text{gn}\) options always imply the \(-\text{mn}\) 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
Figure 15: Snapshot of an OPEN INVENTOR display of a sphere partitioned into 7 almost equal pieces by mapping onto the complete graph with 7 vertices. Vertices of same color are mapped onto the same processor.

is correct. In this case, use the “-” parameter to take standard input as a dummy mapping input stream.

-o format[arguments]
Specify the type of output, with optional parameters within curly braces and separated by commas. The output formats are listed below.

-iview format
Output the graph in SGI's OPEN INVENTOR format, in ASCII mode, suitable for display by the ivview program [40]. 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 format
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.
\[x=\text{val}\]
  Minimum X relative clipping position (in $[0.0;1.0]$).
\[X=\text{val}\]
  Maximum X relative clipping position (in $[0.0;1.0]$).
\[y=\text{val}\]
  Minimum Y relative clipping position (in $[0.0;1.0]$).
\[Y=\text{val}\]
  Maximum Y relative clipping position (in $[0.0;1.0]$).
-\text{V}  Print the program version and copyright.

Default option set is “-Oh{v}”.

### 6.3.13 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**

-\text{-h}  Display the program synopsis.
-\text{-V}  Print the program version and copyright.

### 6.3.14 mcv

**Synopsis**

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

The program `mcv` is the source mesh converter. It takes as 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 [10], 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”.

6.3.15 `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.

6.3.16 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 [17], but classical [39] and state-of-the-art [1, 47] 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 7.3.1 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.

-v 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 5.5. 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 7.3.4.

-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.

-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.

6.3.17 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.
7 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;
- 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 efficiency of Scotch without having to modify their code. The services provided by this library are described in Section 7.15.

7.1 Calling the routines of LIBSCOTCH

7.1.1 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 7.5.2 and 7.5.5, respectively, can be called from C by using the following code.

```c
#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.

7.1.2 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 SCOTCH_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 DOUBLE PRECISIONs, 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 number 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 7.5.2 and 7.5.5, respectively, can be called from Fortran by using the following code.

```
INCLUDE "scotchf.h"
DOUBLE PRECISION GRAFDAT(SCOTCH_GRAPHDIM)
INTEGER RETVAL
...
CALL SCOTCHFGRAPHPHINT (GRAFDAT (1), RETVAL)
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.

7.1.3 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 7.13 for more information.

7.1.4 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 are only available in the C interface.

7.1.5 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 8.3 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 \texttt{INTEGERs}. As for the C interface, the regular \texttt{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 \texttt{INTEGER\*num} type should be used for all graph-related values, in accordance to the size of the \texttt{SCOTCH\_Num} type, as set by the \texttt{"-DINTSIZE=x"} compilation flags. Also, the \texttt{INTEGER\*idx} type represents an integer type of a size equivalent to the one of a \texttt{SCOTCH\_Idx}, as set by the \texttt{"-DIDXSIZE=x"} compilation flags. Values of this type are used in the Fortran interface to represent arbitrary array indices which can span across the whole address space, and consequently deserve special treatment.

In practice, when Scotch is compiled on a 32-bit architecture so as to use 64-bit \texttt{SCOTCH\_Num}s, graph indices should be declared as \texttt{INTEGER\*8}, while error return values should still be declared as plain \texttt{INTEGER} (that is, \texttt{INTEGER\*4}) values. On a 32\_64-bit architecture, irrespective of whether \texttt{SCOTCH\_Num}s are defined as \texttt{INTEGER\*4} or \texttt{INTEGER\*8} quantities, the \texttt{SCOTCH\_Idx} type should always be defined as a 64-bit quantity, that is, an \texttt{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 \texttt{ints} equate 64-bit integers. In this case, there is no need to use any type coercing definition.

Also, the MfIS compatibility library provided by Scotch will not work when \texttt{SCOTCH\_Num}s are not \texttt{ints}, since the interface of MfIS uses regular \texttt{ints} to represent graph indices. In addition to compile-time warnings, an error message will be issued when one of these routines is called.

### 7.2 Data formats

All of the data used in the LibScotch interface are of integer type \texttt{SCOTCH\_Num}. To hide the internals of Scotch to callers, all of the data structures are opaque, that is, declared within \texttt{scotch.h} as dummy arrays of double precision values, for the sake of data alignment. Accessor routines, the names of which end in \texttt{"Size"} and \texttt{"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 \texttt{baseval}, whether \texttt{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.

#### 7.2.1 Architecture format

Target architecture structures are completely opaque. The only way to describe an architecture is by means of a graph passed to the \texttt{SCOTCH\_archBuild} routine.

#### 7.2.2 Graph format

Source graphs are described by means of adjacency lists. The description of a graph requires several \texttt{SCOTCH\_Num} scalars and arrays, as shown in Figures 16 and 17. They
have the following meaning:

**baseval**
Base value for all array indexings.

**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 \( \text{baseval} \leq i < (\text{baseval} + \text{vertnbr}) \), \( \text{vendtab}[i] - \text{verttab}[i] \) is the degree of vertex \( i \), and the indices of the neighbors of \( i \) are stored in edgetab from \( \text{edgetab}[\text{verttab}[i]] \) to \( \text{edgetab}[\text{vendtab}[i] - 1] \), 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 16, vendtab is of size \( \text{vertnbr} + 1 \) and vendtab points to vendtab + 1. This case is referred to as the “compact edge array” case, such that vendtab is sorted in ascending order, vendtab[baseval] = baseval and vendtab[baseval + edgenbr] = (baseval + edgenbr).

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

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

**edlotab**
Optional array, of a size equal at least to \( \max_i(\text{vendtab}[i]) - \text{baseval} \), 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 verttab[i] and vendtab[i] with the values of verttab[vertnbr-1] and vendtab[vertnbr-1], respectively, and browse the adjacencies of all neighbors of former vertex vertnbr-1 such that all (vertnbr-1) indices are turned into is. Then, vertnbr must be decremented, 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.
Figure 16: 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 17: Adjacency structure of the sample graph of Figure 16 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.
To add a new vertex, one has to fill `verttab[vertnbr-1]` and `vendtab[vertnbr-1]` with the starting and end indices of the adjacency sub-array of the new vertex. Then, the adjacencies of its neighbor vertices must also be updated to account for it. If free space had been reserved at the end of each of the neighbors, one just has to increment the `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.

### 7.2.3 Mesh format

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 18. 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 ≤ 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 18, `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.
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 19 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 20. 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.

### 7.2.4 Geometry format

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
Figure 19: 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 20: Re-meshing of the mesh of Figure 19. 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.
as an array of (x), or (x,y), or (x,y,z) tuples, according to **dimnbr.** Co-
ordinates that are not used (e.g. the “z” coordinates for a 2-dimentional
object) are not allocated. Therefore, the “x” coordinate of some graph
vertex \( i \) is located at \( \text{geomtab}[(i - \text{baseval}) * \text{dimnbr} + \text{baseval}] \), its
“y” coordinate is located at \( \text{geomtab}[(i - \text{baseval}) * \text{dimnbr} + \text{baseval} + 1] \) if \( \text{dimnbr} \leq 2 \), and its “z” coordinate is located at \( \text{geomtab}[(i - \text{baseval}) * \text{dimnbr} + \text{baseval} + 2] \) if \( \text{dimnbr} = 3 \). Whenever the ge-
ometry is associated with a mesh, only node vertices are consid ered, so
the “x” coordinate of some mesh node vertex \( i \), with \( \text{vnodbas} \leq i \), is lo-
cated at \( \text{geomtab}[(i - \text{vnodbas}) * \text{dimnbr} + \text{baseval}] \), its “y” coordi-
nate is located at \( \text{geomtab}[(i - \text{vnodbas}) * \text{dimnbr} + \text{baseval} + 1] \) if \( \text{dimnbr} \leq 2 \), and its “z” coordinate is located at \( \text{geomtab}[(i - \text{vnodbas}) * \text{dimnbr} + \text{baseval} + 2] \) if \( \text{dimnbr} = 3 \).

### 7.2.5 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 21. 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 \( \text{vnodbas} > \text{baseval} \). 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
\( i + (\text{vnodbas} - \text{baseval}) \) 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 \( k = \text{permtab}[i] \), 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 \( i = \text{peritab}[k] \), 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 \( \text{baseval} \leq c < \text{(cblknbr + baseval)} \), contains columns with indices ranging from \( \text{rangtab}[i] \) to \( \text{rangtab}[i + 1] \), exclusive, in the reordered matrix. Indices in \( \text{rangtab} \) are based. Therefore, \( \text{rangtab}[(\text{baseval})] \) is always equal to \( \text{baseval} \), and \( \text{rangtab}[(\text{cblknbr} + \text{baseval})] \) is always equal to \( \text{vertnbr} + \text{baseval} \) for
graphs and to \( \text{vnodnbr} + \text{baseval} \) for meshes. In order to avoid memory
errors when column blocks are all single columns, the size of \( \text{rangtab} \) must
always be one more than the number of columns, that is, \( \text{vertnbr} + 1 \) for
graphs and \( \text{vnodnbr} + 1 \) for meshes.

- **treetab**: Array of ascendants of permuted column blocks in the separators tree.
\( \text{treetab}[i] \) is the index of the father of column block \( i \) in the separators
tree, or $-1$ if column block $i$ is the root of the separators tree. Whenever separators or leaves of the separators tree are split into subblocks, as the block splitting, minimum fill or minimum degree methods do, all subblocks of the same level are linked to the column block of higher index belonging to the closest separator ancestor. Indices in \texttt{tree} are based, in the same way as for the other blocking structures. See Figure 21 for a complete example.

\subsection{7.3 Strategy strings}

The behavior of the mapping and block ordering routines of the \textsc{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.

\subsubsection{7.3.1 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 \textsc{libScotch}, which will yield very good results in most cases. By doing so, they will spare themselves the hassle of updating their strategies to comply to subsequent syntactic changes, and they will benefit from the availability of new partitioning or ordering methods as soon as they are made available.

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

The \textsc{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 \texttt{SCOTCH\_strat\*Build}, tune default strategy strings according to parameters provided by the user, such as the requested number of parts (used as a hint to select the most efficient partitioning routines), the desired maximum load imbalance ratio, and a set of preference flags. While some of these flags are antagonistic, most of them can be combined, by means of addition or “binary or” operators. These flags are the following.
SCOTCH_STRATQUALITY
Privilege quality over speed. This is the default behavior of default strategy strings when they are used just after being initialized.

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.

7.3.2 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.

m Multi-level method. The parameters of the multi-level method are listed below.

as=asc
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=low
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 than 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.

type=type
Set the type of matching that is used to coarsen the graphs. type is h for heavy-edge matching, or s for scan (first-fit) matching.

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 $nbr \times \text{domnnbr}$ vertices, where $\text{domnnbr}$ is the number of vertices in the target architecture.

Dual Recursive Bipartitioning mapping algorithm, as defined in section 3.1.3. 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.

### 7.3.3 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 \textit{cond}, either \textit{strat1} or \textit{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.

\textit{cond1} \textsc{\|} \textit{cond2}
Logical or operator. The result of the condition is true if \textit{cond1} or \textit{cond2} are true, or both.

\textit{cond1} \textsc{\&} \textit{cond2}
Logical and operator. The result of the condition is true only if both \textit{cond1} and \textit{cond2} are true.

\textsc{\textsf{^}} \textit{cond}
Logical not operator. The result of the condition is true only if \textit{cond} is false.

\textit{var\,relop\,val}
Relational operator, where \textit{var} is a graph variable, \textit{val} is either a graph variable or a constant of the type of variable \textit{var}, and \textit{relop} is one of ‘<’, ‘=’, and ‘>’. The graph variables are listed below, along with their types.

\textit{deg}
The average degree of the current graph. Float.

\textit{edge}
The number of arcs (which is twice the number of edges) of the current graph. Integer.

\textit{load}
The overall vertex load (weight) of the current graph. Integer.

\textit{load0}
The vertex load of the first subset of the current bipartition of the current graph. Integer.

\textit{vert}
The number of vertices of the current graph. Integer.

\textit{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 [8], was created to reduce the cost of vertex separator refinement algorithms in a multi-level 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.

**Diffusion method.** This method, presented in [42], flows two kinds of antagonistic liquids, scotch and anti-scotch, from two source vertices, and sets the new frontier as the limit between vertices which contain scotch and the ones which contain anti-scotch. Because selecting the source vertices is essential to the obtainment of useful results, this method has been hard-coded so that the two source vertices are the two vertices of highest indices, since in the band method these are the anchor vertices which represent all of the removed vertices of each part. Therefore, this method must be used on band graphs only, or on specifically crafted graphs. Applying it to any other graphs is very likely to lead to extremely poor results. The parameters of the diffusion bipartitioning method are listed below.

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

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

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

**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
multi-level method.

pass=nbr
Set the maximum number of optimization passes performed by the algo-
rithm. 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 num-
ber of passes, that is, as many as needed by the algorithm to converge.

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 Multi-level method. The parameters of the multi-level method are listed be-
low.

asc=strat
Set the strategy that is used to refine the partitions obtained at ascend-
ing 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. Coars-
ening stops when either the coarsening ratio is above the maximum coars-
ening ratio, or the graph has fewer vertices than the minimum number
of vertices allowed.

type=type
Set the type of matching that is used to coarsen the graphs. type is h for
heavy-edge matching, or s for scan (first-fit) matching.
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 3.1.7).

7.3.4 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.

63
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) [29], 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.

c\_\text{min}=\text{size}
Set the minimum size of the resulting subblocks, in number of columns. Blocks larger than twice this minimum size are cut into sub-blocks of equal sizes (within one), having a number of columns comprised between size and 2\_\text{size}.

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.

\text{strat}=\text{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\_\text{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.

\text{rat}=\text{rat}
Set the compression ratio over which graphs and meshes will not be compressed. Useful values range between 0.7 and 0.8.

\text{cpr}=\text{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.

\text{unc}=\text{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 [47]. 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=\textit{size}
Minimum number of columns per column block. All column blocks of
width smaller than \textit{size} are amalgamated to their parent column block in
the elimination tree, provided that it does not violate the \textit{cmax} constraint.

\textit{cmax}=\textit{size}
Maximum number of column blocks over which some column block will
not amalgamate one of its descendants 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.

\textit{frat}=\textit{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.

\textit{cmin}=\textit{size}
Minimum number of columns per column block. All column blocks of
width smaller than \textit{size} are amalgamated to their parent column block in
the elimination tree, provided that it does not violate the \textit{cmax} constraint.

\textit{cmax}=\textit{size}
Maximum number of column blocks over which some column block will
not amalgamate one of its descendants 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.

\textit{frat}=\textit{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 \textit{s} method should be preferred. This method
has only one parameter.

\textit{pass}=\textit{nbr}
Set the number of sweeps performed by the algorithm.

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

\textit{ole}=\textit{strat}
Set the ordering strategy that is used on every leaf of the separators tree
if the node separation strategy \textit{sep} has failed to separate it further.

\textit{ose}=\textit{strat}
Set the ordering strategy that is used on every separator of the separators
tree.
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 7.3.5.

**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.

**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.

**Graph ordering strategy to apply to the associated graph.**

### 7.3.5 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.

\neg 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.

\textbf{levl}
The level of the subgraph in the separators tree, starting from zero at the root of the tree. Integer.

\textbf{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.

\textbf{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.

\textbf{vert}
The number of vertices of the current subgraph. Integer.

The currently available vertex separation methods are the following.

\textbf{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 [8], was created to reduce the cost of separator refinement algorithms in a multi-level context, but it improves partition quality too. The parameters of the band separation method are listed below.

\textbf{bnd=}\textit{strat}
Set the vertex separation strategy to be used on the band graph.

\textbf{org=}\textit{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 [48], which uses a variant of the Hopcroft and Karp algorithm due to Duff [9]. This method is expensive and most often yields poorer results than direct vertex-oriented methods such as the 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 within section 7.3.3, at page 59.

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.

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

asc=strat
Set the strategy that is used to refine the 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 multi-level 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.
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_\text{nnnnnn}.map”, where “\text{nnnnnn}” 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.

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

7.4 Target architecture handling routines

7.4.1 SCOTCH_archInit

Synopsis

\begin{verbatim}
int SCOTCH_archInit (SCOTCH_Arch * archptr)

scotchfarchinit (doubleprecision (*)(*) archdat, integer ierr)
\end{verbatim}

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 graph structure has been successfully initialized, and 1 else.

7.4.2 SCOTCH_archExit

Synopsis

\begin{verbatim}
void SCOTCH_archExit (SCOTCH_Arch * archptr)

scotchfarchexit (doubleprecision (*)(*) archdat)
\end{verbatim}

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.
7.4.3 SCOTCH_archLoad

Synopsis

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

```c
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 5.4).

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.

7.4.4 SCOTCH_archSave

Synopsis

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

```c
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 5.4).

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.
7.4.5 SCOTCH_archBuild

Synopsis

```c
int SCOTCH_archBuild (SCOTCH_Arch * archptr,
                      const SCOTCH_Graph * grafptr,
                      const SCOTCH_Num      listnbr,
                      const SCOTCH_Num *    listtab,
                      const SCOTCH_Strat *  straptr)
```

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

Description

The SCOTCH_archBuild routine fills the architecture structure pointed to by `archptr` with the 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 111. The syntax of bipartitioning strategy strings is defined in section 7.3.2, page 59. Additional information may be obtained from the manual page of `amk_grf`, the stand-alone executable that uses function SCOTCH_archBuild to build decomposition-defined target architecture from source graphs, available at page 32.

Return values

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

7.4.6 SCOTCH_archCmplt

Synopsis

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

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

### 7.4.7 `SCOTCH_archCmpltw`

**Synopsis**

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

```c
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.

### 7.4.8 `SCOTCH_archName`

**Synopsis**

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

```c
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 null 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.

7.4.9 SCOTCH_archSize

Synopsis

```c
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.

7.4.10 SCOTCH_archHcub

Synopsis

```c
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 of dimension `hdimval`.

Return values

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

74
7.4.11 SCOTCH_archMesh2D

Synopsis

```c
int SCOTCH_archMesh2D (SCOTCH_Arch * archptr,
                       const SCOTCH_Num xdimval,
                       const SCOTCH_Num ydimval)
```

```c
scotchfarchmesh2d (doubleprecision (*) archdat,
                   integer* num xdimval,
                   integer* num ydimval,
                   integer ierr)
```

Description

The SCOTCH_archMesh2D 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_archMesh2D returns 0 if the 2D mesh target architecture has been successfully built, and 1 else.

7.4.12 SCOTCH_archMesh3D

Synopsis

```c
int SCOTCH_archMesh3D (SCOTCH_Arch * archptr,
                       const SCOTCH_Num xdimval,
                       const SCOTCH_Num ydimval,
                       const SCOTCH_Num zdimval)
```

```c
scotchfarchmesh3d (doubleprecision (*) archdat,
                   integer* num xdimval,
                   integer* num ydimval,
                   integer* num zdimval,
                   integer ierr)
```

Description

The SCOTCH_archMesh3D 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_archMesh3D returns 0 if the 3D mesh target architecture has been successfully built, and 1 else.
7.4.13  SCOTCH_archTleaf

Synopsis

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

```c
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 \( \sum_{i=0}^{levlnbr-1} \text{sizetab}[i] \) processors. Level 0 is the root of the tree. For each level \( i \), with \( 0 \leq i < levlnbr \), `sizetab[i]` is the number of child nodes 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 5.4.2, page 23, for an example of such an architecture.

Return values

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

7.4.14  SCOTCH_archTorus2D

Synopsis

```c
int SCOTCH_archTorus2D (SCOTCH_Arch * archptr,
                        const SCOTCH_Num xdimval,
                        const SCOTCH_Num ydimval)
```

```c
scotchfarchtorus2d (doubleprecision (*) archdat,
                    integer num xdimval,
                    integer num ydimval,
                    integer ierr)
```

Description

The `SCOTCH_archTorus2D` 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_archTorus2D` returns 0 if the 2D torus target architecture has been successfully built, and 1 else.
7.4.15  SCOTCH_archTorus3D

Synopsis

```c
int SCOTCH_archTorus3D (SCOTCH_Arch * archptr,
          const SCOTCH_Num xdimval,
          const SCOTCH_Num ydimval,
          const SCOTCH_Num zdimval)
scotchfarchtorus3d (doubleprecision (*) archdat,
          integer* num xdimval,
          integer* num ydimval,
          integer* num zdimval,
          integer ierr)
```

Description

The SCOTCH_archTorus3D 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_archTorus3D returns 0 if the 3D torus target architecture has been successfully built, and 1 else.

7.5  Graph handling routines

7.5.1  SCOTCH_graphAlloc

Synopsis

```c
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. 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.

7.5.2  SCOTCH_graphInit

Synopsis

```c
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.

7.5.3 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.

7.5.4 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_graphExit 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.
7.5.5 SCOTCH::graphLoad

Synopsis

```c
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 5.1).

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.

7.5.6 SCOTCH::graphSave

Synopsis

```c
int SCOTCH::graphSave (const SCOTCH::Graph * grafptr,
                        FILE * stream)

scotchfgraphsSave (doubleprecision (*) grafdat,
                   integer fildes,
                   integer ierr)
```
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 5.1). Fortran users must use the `PXFFILENO` or `FNUM` functions to obtain the number of the Unix file descriptor `fildes` associated with the logical unit of the graph file.

**Return values**

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

### 7.5.7 SCOTCH_graphBuild

**Synopsis**

```c
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)
```

```c
scotchfgraphbuild (doubleprecsion (*) 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.

7.5.8 SCOTCH_graphBase

Synopsis

```c
int SCOTCH_graphBase (SCOTCH_Graph * graptr, 
                      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.
### 7.5.9 SCOTCH_graphCheck

**Synopsis**

```c
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 user-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.

### 7.5.10 SCOTCH_graphSize

**Synopsis**

```c
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.
7.5.11 SCOTCH_graphData

Synopsis

```c
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);
```

```c
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 edlidx);
```

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 `vertnbr`. `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 7.1.5 for more information on this issue.

### 7.5.12 SCOTCH\_graphStat

**Synopsis**

```c
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)
```
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.

### 7.6 Graph mapping and partitioning routines

The first two routines provide high-level functionalities and free the user from the burden of calling in sequence several of the low-level routines described afterward.

#### 7.6.1 SCOTCH_graphPart

**Synopsis**

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

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

**Description**

The SCOTCH_graphPart routine provides high-level functionalities and frees the user from the burden of calling in sequence several of the low-level routines described afterward.
The SCOTCH_graphPart routine computes a partition into partnbr parts of the source graph structure pointed to by grafptr, using the graph partitioning strategy pointed to by stratptr, 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. Parts are numbered from 0 to partnbr − 1.

Return values

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

7.6.2 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, 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.

Return values

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

Synopsis

```c
int SCOTCH_graphMapInit (const SCOTCH_Graph * grafptr,
                         SCOTCH_Mapping * mappptr,
                         const SCOTCH_Arch * archptr,
                         SCOTCH_Num * parttab)
```

```c
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_Num 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_graphMapExit to free its internal structures.

Return values

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

7.6.4 SCOTCH_graphMapExit

Synopsis

```c
void SCOTCH_graphMapExit (const SCOTCH_Graph * grafptr,
                          SCOTCH_Mapping * mappptr)
```

```c
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.
7.6.5 SCOTCH_graphMapLoad

Synopsis

```c
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 5.5) 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 mapping file.

Return values

`SCOTCH_graphMapLoad` returns 0 if the mapping structure has been successfully loaded from `stream`, and 1 else.

7.6.6 SCOTCH_graphMapSave

Synopsis

```c
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 5.5).

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.
7.6.7 SCOTCH_graphMapCompute

Synopsis

```c
int SCOTCH_graphMapCompute (const SCOTCH_Graph * grafptr,
    SCOTCH_Mapping * mappptr,
    const SCOTCH_Strat * straptr)
```

```c
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 (see section 7.6.3) 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 content is not significant.

7.6.8 SCOTCH_graphMapView

Synopsis

```c
int SCOTCH_graphMapView (const SCOTCH_Graph * grafptr,
    const SCOTCH_Mapping * mappptr,
    FILE * stream)
```

```c
scotchfgraphmapview (doubleprecision (*)(*) grafdat,
    doubleprecision (*)(*) mappdat,
    integer fildes,
    integer ierr)
```

Description

The SCOTCH_mapView 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_mapView returns 0 if the data has been successfully written to stream, and 1 else.

7.7 Graph ordering routines

The first routine provides high-level functionality and frees the user from the burden of calling in sequence several of the low-level routines described afterward.

7.7.1 SCOTCH_graphOrder

Synopsis

```c
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)
```

```fortran
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
\( \text{vertnbr} - 1 \) if the graph base is 0, and from 1 to \( \text{vertnbr} \) 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 7.2.5 for more information.

**Return values**

\text{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.

### 7.7.2 SCOTCH\_graphOrderInit

**Synopsis**

```c
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)
```

```c
scotchfgraphorderinit (doubleprecision (*) grafdat,
                      doubleprecision (*) ordedat,
                      integer*num (*) permtab,
                      integer*num (*) peritab,
                      integer*num (*cblkptr,
                      integer*num (*rangtab,
                      integer*num (*treetab,
                      integer ierr)
```

**Description**

The \text{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 \text{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 7.2.5, 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 free its internal structures.

Return values

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

7.7.3 SCOTCH_graphOrderExit

Synopsis

```c
void SCOTCH_graphOrderExit (const SCOTCH_Graph * grafptr,
                           SCOTCH_Ordering * ordeptr);
```

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.

7.7.4 SCOTCH_graphOrderLoad

Synopsis

```c
int SCOTCH_graphOrderLoad (const SCOTCH_Graph * grafptr,
                            SCOTCH_Ordering * ordeptr,
                            FILE * stream);
```

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 5.6) 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.

7.7.5 SCOTCH_graphOrderSave

Synopsis

```c
int SCOTCH_graphOrderSave (const SCOTCH_Graph * grafptr,
                           const SCOTCH_Ordering * ordeptr,
                           FILE * stream)
```

```c
scotchf_graphordersave (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, in the SCOTCH ordering format (see section 5.6).

Fortran users must use the PXFILENO 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.

7.7.6 SCOTCH_graphOrderSaveMap

Synopsis

```c
int SCOTCH_graphOrderSaveMap (const SCOTCH_Graph * grafptr,
                              const SCOTCH_Ordering * ordeptr,
                              FILE * stream)
```

```c
scotchf_graphordersavemap (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, in the SCOTCH mapping format (see section 5.5). 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 6.3.12) 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.

### 7.7.7 SCOTCH_graphOrderSaveTree

**Synopsis**

```c
int SCOTCH_graphOrderSaveTree (const SCOTCH_Graph * grafptr,  
const SCOTCH_Ordering * ordeptr, 
FILE * stream)
```

**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.

### 7.7.8 SCOTCH_graphOrderCheck

**Synopsis**

```c
int SCOTCH_graphOrderCheck (const SCOTCH_Graph * grafptr,  
const SCOTCH_Ordering * ordeptr)
```

**Description**

The `SCOTCH_graphOrderCheck` routine checks the tree hierarchy information associated with the `SCOTCH_Ordering` structure pointed to by `ordeptr`.

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_graphOrderCheck` returns 0 if the separators tree structure has been successfully written to `stream`, and 1 else.
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.

### 7.7.9 SCOTCH_graphOrderCompute

**Synopsis**

```c
int SCOTCH_graphOrderCompute (const SCOTCH_Graph * grafptr,
                               SCOTCH_Ordering * ordeptr,
                               const SCOTCH_Strat * straptr)
```

```c
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 7.7.2 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.

### 7.7.10 SCOTCH_graphOrderComputeList

**Synopsis**

```c
int SCOTCH_graphOrderComputeList (const SCOTCH_Graph * grafptr,
                                  SCOTCH_Ordering * ordeptr,
                                  SCOTCH_Num * listnbr,
                                  SCOTCH_Num * listtab,
                                  const SCOTCH_Strat * straptr)
```

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

95
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 7.2.5 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 3.2.3 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.

7.8 Mesh handling routines

7.8.1 SCOTCH_meshAlloc

Synopsis

SCOTCHMesh * 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. 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.

7.8.2 SCOTCH meshInit

Synopsis

```c
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.

7.8.3 SCOTCH meshExit

Synopsis

```c
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.

7.8.4 SCOTCH meshLoad

Synopsis

```c
int SCOTCH_meshLoad (SCOTCH_Mesh * meshptr,
                      FILE * stream,
                      SCOTCH_Num baseval)
scotchfmeshload (doubleprecision (*) meshdat,
                 integer fildes,
                 integer*num baseval,
                 integer ierr)
```

97
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 5.2).

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.

7.8.5 SCOTCH_meshSave

Synopsis

```c
int SCOTCH_meshSave (const SCOTCH.Mesh * meshptr, 
                     FILE * stream)
```

```fortran
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 5.2).

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.

7.8.6 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 = velmbr 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 (velmbas + vnodnbr + 1) if the edge array is compact (that is, if vendtab equals vendtab + 1 or NULL), or of size (velmbas + vnodnbr) else. vendtab is the adjacency end index array, of size (velmbas + vnodnbr) if it is disjoint from verttab. velotab is the element vertex load array, of size velmbr if it exists. vnlotab is the node vertex load array, of size vnodnbr if it exists. vlbltab is the vertex label array, of size velmbr + 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.

7.8.7 SCOTCH_meshCheck

Synopsis

```c
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.

7.8.8 SCOTCH_meshSize

Synopsis

```c
void SCOTCH_meshSize (const SCOTCH_Mesh * meshptr,
                      SCOTCH_Num * velmptr,
                      SCOTCH_Num * vnodptr,
                      SCOTCH_Num * edgeptr)
```

100
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.

7.8.9 SCOTCH_meshData

Synopsis

void SCOTCH_meshData (const SCOTCH_Mesh * meshptr, 
  SCOTCH_Num * vebaptr, 
  SCOTCH_Num * vnbatr, 
  SCOTCH_Num * velmptr, 
  SCOTCH_Num * vnodptr, 
  SCOTCH_Num ** verttab, 
  SCOTCH_Num ** vendtab, 
  SCOTCH_Num ** velotab, 
  SCOTCH_Num ** vnlotab, 
  SCOTCH_Num ** vlbtab, 
  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 vnodnbnr, 
  integer*idx vertidx, 
  integer*idx vendidx, 
  integer*idx veloidx, 
  integer*idx vnloidx, 
  integer*idx vlbldx, 
  integer*num edgenbr, 
  integer*idx edgeidx, 
  integer*num degrmax)
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 vnabaptr 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 reference to the adjacency array, of size at least edgenbr. degptr 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 3264 architectures, such indices can be larger than the size of a regular INTEGER. This is why the indices to be returned are defined by means of a specific integer type. See Section 7.1.5 for more information on this issue.

### 7.8.10 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 vnlodltd,
    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 element vertex degree, the maximum element vertex degree, the average element vertex degree, and the variance of the element vertex degrees, respectively.

7.8.11 SCOTCH_meshGraph

Synopsis

int SCOTCH_meshGraph (const SCOTCH_Mesh * meshptr,
                      SCOTCH_Graph * grafptr)
The \texttt{SCOTCH\_meshGraph} routine builds a graph from a mesh. It creates in the \texttt{SCOTCH\_Graph} structure pointed to by \texttt{grafptr} a graph having as many vertices as there are nodes in the \texttt{SCOTCH\_Mesh} structure pointed to by \texttt{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 \texttt{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.

\textbf{Return values}

\texttt{SCOTCH\_meshGraph} returns 0 if the graph structure has been successfully allocated and filled, and 1 else.

\section*{7.9 Mesh ordering routines}

The first routine provides high-level functionality and frees the user from the burden of calling in sequence several of the low-level routines described afterward.

\subsection*{7.9.1 \texttt{SCOTCH\_meshOrder}}

\textbf{Synopsis}

\begin{verbatim}
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)
\end{verbatim}
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 7.2.5 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.

7.9.2 SCOTCH\_meshOrderInit

Synopsis
The \texttt{SCOTCH\_meshOrderInit} routine fills the ordering structure pointed to by \texttt{ordeptr} with all of the data that are passed to it. Thus, all subsequent calls to ordering routines such as \texttt{SCOTCH\_meshOrderCompute}, using this ordering structure as parameter, will place ordering results in fields \texttt{permtab}, \texttt{peritab}, \texttt{cblkptr}, \texttt{rangtab} or \texttt{treetab}, if they are not set to \texttt{NULL}.

\texttt{permtab} is the ordering permutation array, of size \texttt{vnodnbr}, \texttt{peritab} is the inverse ordering permutation array, of size \texttt{vnodnbr}, \texttt{cblkptr} is the pointer to a \texttt{SCOTCH\_Num} that will receive the number of produced column blocks, \texttt{rangtab} is the array that holds the column block span information, of size \texttt{vnodnbr} + 1, and \texttt{treetab} is the array holding the structure of the separators tree, of size \texttt{vnodnbr}. See the above manual page of \texttt{SCOTCH\_meshOrder}, as well as section 7.2.5, for an explanation of the semantics of all of these fields.

The \texttt{SCOTCH\_meshOrderInit} routine should be the first function to be called upon a \texttt{SCOTCH\_Ordering} structure for ordering meshes. When the ordering structure is no longer of use, the \texttt{SCOTCH\_meshOrderExit} function must be called, in order to free its internal structures.

### Return values

\texttt{SCOTCH\_meshOrderInit} returns 0 if the ordering structure has been successfully initialized, and 1 else.

### 7.9.3 \texttt{SCOTCH\_meshOrderExit}

### Synopsis

```c
void SCOTCH\_meshOrderExit (const SCOTCH\_Mesh * meshptr,
                           SCOTCH\_Ordering * ordeptr)
scotchfmeshorderexit (doubleprecision (*) meshdat,
doubleprecision (*) ordedat)
```
Description

The \texttt{SCOTCH\_meshOrderExit} function frees the contents of a \texttt{SCOTCH\_Ordering} structure previously initialized by \texttt{SCOTCH\_meshOrderInit}. All subsequent calls to \texttt{SCOTCH\_meshOrder\*} routines other than \texttt{SCOTCH\_meshOrderInit}, using this structure as parameter, may yield unpredictable results.

7.9.4 \texttt{SCOTCH\_meshOrderSave}

Synopsis

\begin{verbatim}
int SCOTCH\_meshOrderSave (const SCOTCH\_Mesh * meshptr,  
const SCOTCH\_Ordering * ordeptr,  
FILE  * stream)

cotchfmeshordersave (doubleprecision (*) meshdat,  
doubleprecision (*) ordesdat,  
integer  fildes,  
integer  ierr)
\end{verbatim}

Description

The \texttt{SCOTCH\_meshOrderSave} routine saves the contents of the \texttt{SCOTCH\_Ordering} structure pointed to by \texttt{ordeptr} to stream \texttt{stream}, in the \texttt{SCOTCH} ordering format (see section 5.6).

Return values

\texttt{SCOTCH\_meshOrderSave} returns 0 if the ordering structure has been successfully written to \texttt{stream}, and 1 else.

7.9.5 \texttt{SCOTCH\_meshOrderSaveMap}

Synopsis

\begin{verbatim}
int SCOTCH\_meshOrderSaveMap (const SCOTCH\_Mesh * meshptr,  
const SCOTCH\_Ordering * ordeptr,  
FILE  * stream)

cotchfmeshordersavemap (doubleprecision (*) meshdat,  
doubleprecision (*) ordesdat,  
integer  fildes,  
integer  ierr)
\end{verbatim}

Description

The \texttt{SCOTCH\_meshOrderSaveMap} routine saves the block partitioning data associated with the \texttt{SCOTCH\_Ordering} structure pointed to by \texttt{ordeptr} to stream \texttt{stream}, in the \texttt{SCOTCH} mapping format (see section 5.5). 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 6.3.12) 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 6.3.8).

**Return values**

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

### 7.9.6 SCOTCH_meshOrderSaveTree

**Synopsis**

```c
int SCOTCH_meshOrderSaveTree (const SCOTCH_Mesh * meshptr, const SCOTCH_Ordering * ordeptr, FILE * stream)
```

**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.

### 7.9.7 SCOTCH_meshOrderCheck

**Synopsis**

```c
int SCOTCH_meshOrderCheck (const SCOTCH_Mesh * meshptr, const SCOTCH_Ordering * ordeptr)
```
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.

7.9.8 SCOTCH_meshOrderCompute

Synopsis

int SCOTCH_meshOrderCompute (const SCOTCH_Mesh * meshptr,
                           SCOTCH_Ordering * ordeptr,
                           const SCOTCH_Strat * stratptr)

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 meshptr, 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 7.9.2 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.

7.10 Strategy handling routines
7.10.1 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. 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.

### 7.10.2 SCOTCH_stratInit

**Synopsis**

```c
int SCOTCH_stratInit (SCOTCH_Strat * straptr)
scotchfstratinit (doubleprecision (*) stradat,
    integer             ierr)
```

**Description**

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

**Return values**

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

### 7.10.3 SCOTCH_stratExit

**Synopsis**

```c
void SCOTCH_stratExit (SCOTCH_Strat * archptr)
sotchfstratexit (doubleprecision (*) stradat)
```

**Description**

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

### 7.10.4 SCOTCH_stratSave

**Synopsis**

```c
int SCOTCH_stratSave (const SCOTCH_Strat * strapttr,
    FILE *                      stream)
```
scotchfstratsave (doubleprecision (*) stradat,
    integer fildes,
    integer ierr)

Description

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

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

Return values

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

7.10.5 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.

7.10.6 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.

7.10.7 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 prefe-
rence 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_graph
Map, for instance. See Section 7.3.1 for a description of the available flags.

Return values

SCOTCH_stratGraphMapBuild returns 0 if the strategy string has been suc-
cessfully set, and 1 else.
7.10.8 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 \texttt{straptr} with the graph ordering strategy string pointed to by \texttt{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 \texttt{string} must be null-terminated.

Return values

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

7.10.9 SCOTCH_stratGraphOrderBuild

Synopsis

    int SCOTCH_stratGraphOrderBuild (SCOTCH_Strat * straptr,
                                     const SCOTCH_Num flagval,
                                     const double balrat)

scotchfstratgraphorderbuild (doubleprecision (*) stradat,
                           integer* num flagval,
                           doubleprecision balrat,
                           integer ierr)

Description

The SCOTCH_stratGraphOrderBuild routine fills the strategy structure pointed to by \texttt{straptr} with a default ordering strategy tuned according to the preference flags passed as \texttt{flagval} and to the desired nested dissection imbalance ratio \texttt{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 7.3.1 for a description of the available flags.

Return values

SCOTCH_stratGraphOrderBuild returns 0 if the strategy string has been successfully set, and 1 else.
7.10.10 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.

7.10.11 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_meshOrder, for instance. See Section 7.3.1 for a description of the available flags.

Return values

SCOTCH_stratMeshOrderBuild returns 0 if the strategy string has been successfully set, and 1 else.
7.11 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_10 structures.

Apart from the routines that create, destroy or access SCOTCH_10 structures, all of the routines in this section are input/output routines, which read or write both SCOTCH_10Graph and SCOTCH_10 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.

7.11.1 SCOTCH_geomAlloc

Synopsis

```
SCOTCHGeom * SCOTCH_geomAlloc (void)
```

Description

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

Return values

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

7.11.2 SCOTCH_geomInit

Synopsis

```
int SCOTCH_geomInit (SCOTCHGeom * geomptr)
    scotchfgeominit (doubleprecision (* geomdat, integer ierr)
```

Description

The SCOTCH_geomInit function initializes a SCOTCHGeom 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.

7.11.3 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.

7.11.4 SCOTCH_geomData

Synopsis

void SCOTCH_geomData (const SCOTCH_Geom * geomptr,
SCOTCH_Num * dimnptr,
double ** geomtab)
scotchfgeomdata (doubleprecision (*) geomdat,
doubleprecision (*) indxtab,
integer*num dimnabr,
integer*idx geomidx)

Description

The SCOTCH_geomData routine is a multiple accessor to the contents of SCOTCH_Geom structures.

dimmnptr 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 7.2.4.

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 \texttt{myarray(1)} is passed as parameter \texttt{indxtab}, then the first cell of array \texttt{geomtab} will be accessible as \texttt{myarray(geomidx)}. In order for this feature to behave properly, the \texttt{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 \texttt{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 \texttt{INTEGER}. This is why the indices to be returned are defined by means of a specific integer type. See Section 7.1.5 for more information on this issue.

7.11.5 SCOTCH\_graphGeomLoadChac

Synopsis

```c
int SCOTCH\_graphGeomLoadChac (SCOTCH\_Graph * grafptr,
                                SCOTCH\_Geom * geomptr,
                                FILE * grafstream,
                                FILE * geomstream,
                                const char * string)
```

Description

The \texttt{SCOTCH\_graphGeomLoadChac} routine fills the \texttt{SCOTCH\_Graph} structure pointed to by \texttt{grafptr} with the source graph description available from stream \texttt{grafstream} in the CHACO graph format [24]. Since this graph format does not handle geometry data, the \texttt{geomptr} and \texttt{geomstream} fields are not used, as well as the \texttt{string} field.

Fortran users must use the \texttt{PXFFILENO} or \texttt{FNUM} functions to obtain the number of the Unix file descriptor \texttt{graffildes} associated with the logical unit of the graph file.

Return values

\texttt{SCOTCH\_graphGeomLoadChac} returns 0 if the graph structure has been successfully allocated and filled with the data read, and 1 else.

7.11.6 SCOTCH\_graphGeomSaveChac

Synopsis

```c
```
The `SCOTCH_graphGeomSaveChac` routine saves the contents of the `SCOTCH_Graph` structure pointed to by `grafptr` to stream `grafstream`, in the Chaco graph format [24]. 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.

### 7.11.7 SCOTCH_graphGeomLoadHabo

**Synopsis**

```c
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 [10]. 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.

7.11.8 SCOTCH_graphGeomLoadScot

Synopsis

```c
int SCOTCH_graphGeomLoadScot (SCOTCH_Graph * grafptr,
                               SCOTCH_Geom  * geomptr,
                               FILE       * grafstream,
                               FILE       * geomstream,
                               const char  * 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 grafstream and geomstream in the SCOTCH graph and geometry formats (see sections 5.1 and 5.3, 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.

7.11.9 SCOTCH_graphGeomSaveScot

Synopsis

```c
int SCOTCH_graphGeomSaveScot (const SCOTCH_Graph * grafptr,
                               const SCOTCH_Geom  * geomptr,
                               FILE       * grafstream,
                               FILE       * geomstream,
                               const char  * string)
```
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 5.1 and 5.3, 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.

7.11.10 SCOTCH_meshGeomLoadHabo

Synopsis

```c
int SCOTCH_meshGeomLoadHabo (SCOTCH_Mesh * meshptr,
                    SCOTCH_Geom * geomptr,
                    FILE * meshstream,
                    FILE * geomstream,
                    const char * string)
```

```c
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 [10]. 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.

7.11.11 SCOTCH_meshGeomLoadScot

Synopsis

```c
int SCOTCH_meshGeomLoadScot (SCOTCH_Mesh * meshptr,
   SCOTCH_Geom * geomptr,
   FILE * meshstream,
   FILE * geomstream,
   const char * string)
```

```fortran
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 5.2 and 5.3, respectively). The string field is not used.

Fortran users must use the PXFILENO 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.

7.11.12 SCOTCH_meshGeomSaveScot

Synopsis

```c
int SCOTCH_meshGeomSaveScot (const SCOTCH_Mesh * meshptr,
   const SCOTCH_Geom * geomptr,
   FILE * meshstream,
   FILE * geomstream,
   const char * string)
```

```fortran
scotchfmeshgeomsavescot (doubleprecision (*) meshdat,
                        doubleprecision (*) geomdat,
                        integer meshfildes,
                        integer geomfildes,
                        character (*) string)
```

121
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 5.2 and 5.3, 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.

7.12 Other data structure routines

7.12.1 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.

Return values

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

7.12.2 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.

Return values

SCOTCH_orderAlloc returns the pointer to the memory area if it has been successfully allocated, and NULL else.
7.13 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.

7.13.1 SCOTCH_errorPrint

Synopsis

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

Description

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

7.13.2 SCOTCH_errorPrintW

Synopsis

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

Description

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

7.13.3 SCOTCH_errorProg

Synopsis

    void SCOTCH_errorProg (const char * progstr)

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

7.14 Miscellaneous routines

7.14.1 SCOTCH_randomReset

Synopsis

    void SCOTCH_randomReset (void)
    scotchfrandomreset ()

Description

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

7.14.2 SCOTCH_version

Synopsis

    int SCOTCH_version (int * const versptr,
        int * const relaptr,
        int * const 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.

7.15 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 9 for an example.

7.15.1 METIS_EdgeND

Synopsis

```
void METIS_EdgeND (const int * const n,
                   const int * const xadj,
                   const int * const adjncy,
                   const int * const numflag,
                   const int * const options,
                   int * const perm,
                   int * const 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. The options array is not used. 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.

7.15.2 METIS_NodeND

Synopsis
void METIS_NodeND (const int * const n,
    const int * const xadj,
    const int * const adjncy,
    const int * const numflag,
    const int * const options,
    int * const perm,
    int * const iperm)

metis_nodend (integer n,
    integer (*) xadj,
    integer (*) adjncy,
    integer numflag,
    integer (*) options,
    integer (*) perm,
    integer (*) 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. The options array is not used. 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.

7.15.3 METIS_NodeWND

Synopsis

void METIS_NodeWND (const int * const n,
    const int * const xadj,
    const int * const adjncy,
    const int * const vwgt,
    const int * const numflag,
    const int * const options,
    int * const perm,
    int * const iperm)

metis_nodwend (integer n,
    integer (*) xadj,
    integer (*) adjncy,
    integer (*) vwgt,
    integer numflag,
    integer (*) options,
    integer (*) perm,
    integer (*) iperm)

126
The **METIS** function performs a nested dissection ordering of the graph passed as arrays `xadj`, `adjncy` and `vwgt`, using the default **SCOTCH** ordering strategy. The `options` array is not used. 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** call the same **SCOTCH** routine, which uses the **SCOTCH** default ordering strategy proved to be efficient in most cases.

### 7.15.4 **METIS** PartGraphKway

**Synopsis**

```c
void METIS_PartGraphKway (const int * const n,
const int * const xadj,
const int * const adjncy,
const int * const vwgt,
const int * const adjwgt,
const int * const wgtflag,
const int * const numflag,
const int * const nparts,
int * const options,
int * const edgecut,
int * const part)
```

```c
metis_partgraphkway (integer n,
integer (*) xadj,
integer (*) adjncy,
integer (*) vwgt,
integer (*) adjwgt,
integer wgtflag,
integer numflag,
integer nparts,
integer (*) options,
integer edgecut,
integer (*) part)
```

**Description**

The **METIS** 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. The `options` array is not used. The `part` array has the same meaning as the `parttab` array of **SCOTCH**. All of the three **METIS** stubs **METIS** call the same **SCOTCH** routine, which
uses the SCOTCH default mapping strategy proved to be efficient in most cases.

### 7.15.5 METIS_PartGraphRecursive

**Synopsis**

```c
void METIS_PartGraphRecursive (const int * const n,
    const int * const xadj,
    const int * const adjncy,
    const int * const vwgt,
    const int * const adjwgt,
    const int * const wgtflag,
    const int * const numflag,
    const int * const nparts,
    const int * const options,
    int * const edgecut,
    int * const part)
```

```c
metis_partgraphrecursive (integer n,
    integer (*) xadj,
    integer (*) adjncy,
    integer (*) vwgt,
    integer (*) adjwgt,
    integer wgtflag,
    integer numflag,
    integer nparts,
    integer (*) options,
    integer edgecut,
    integer (*) 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. The `options` array is not used. 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 MexIS stubs `METIS_PartGraphKway`, `METIS_PartGraphRecursive` and `METIS_PartGraphVKway` call the same SCOTCH routine, which uses the SCOTCH default mapping strategy proved to be efficient in most cases.

### 7.15.6 METIS_PartGraphVKway

**Synopsis**

128
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. The options array is not used. The part array has the same meaning as the parttab array of Scotch.

Since Scotch does not have methods for explicitly 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 \(mbox{vsize}(u) + mbox{vsize}(v)\). 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 7.2.3).

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 Installation

Version 5.1 of the Scotch software package is distributed as free/libre software under the CeCILL-C free/libre software license [6], 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

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 scotch5.1.11 directory, containing several subdirectories and files. Please refer to the files called LICENSE_EN.txt or LICENCE_FR.txt, as well as file INSTALL_EN.txt, to see under which conditions your distribution of Scotch is licensed and how to install it.

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

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

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

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

Whenever doing so, make sure to use integer types of equivalent length to declare variables passed to Scotch routines from caller C and Fortran procedures. Also, because of API conflicts, the MeBIS 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 7.5.11, 7.8.9 and 7.11.4). For 32,64 architectures, such indices can be larger than the
size of a regular \texttt{INTEGER}. This is why the indices to be returned are defined by means of a specific integer type, \texttt{SCOTCH\_Idx}. To coerce the length of this index type to 32 or 64 bits, one can use the \enquote{-DIDX\_SIZE32} or \enquote{-DIDX\_SIZE64} flags, respectively, or else use the \enquote{-DIDX=} definition, at compile time. For instance, adding \enquote{-DIDX=\texttt{long long}} to the \texttt{CFLAGS} variable in the \texttt{Makefile.inc} file to be placed at the root of the source tree will equate all \texttt{SCOTCH\_Idx} integers to \texttt{C long long} integers. By default, when the size of \texttt{SCOTCH\_Idx} is not explicitly defined, it is assumed to be the same as the size of \texttt{SCOTCH\_Num}.

9 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 \enquote{scotch.5.1} of the Scotch distribution. Character \enquote{\%} represents the shell prompt.

- Partition source graph \texttt{brol.grf} into 7 parts, and save the result to file \texttt{/tmp/brol.map}.

\begin{verbatim}
% echo cmplt 7 > /tmp/k7.tgt
% gmap brol.grf /tmp/k7.tgt /tmp/brol.map
\end{verbatim}

This can also be done in a single piped command:

\begin{verbatim}
% echo cmplt 7 | gmap brol.grf - /tmp/brol.map
\end{verbatim}

If compressed data handling is enabled, read the graph as a \texttt{gzip} compressed file, and output the mapping as a \texttt{bzip2} file, on the fly:

\begin{verbatim}
% echo cmplt 7 | gmap brol.grf.gz - /tmp/brol.map.bz2
\end{verbatim}

- Partition source graph \texttt{brol.grf} into two uneven parts of respective weights \(\frac{4}{11}\) and \(\frac{7}{11}\), and save the result to file \texttt{/tmp/brol.map}.

\begin{verbatim}
% echo cmpltw 2 4 7 > /tmp/k2w.tgt
% gmap brol.grf /tmp/k2w.tgt /tmp/brol.map
\end{verbatim}

This can also be done in a single piped command:

\begin{verbatim}
% echo cmpltw 2 4 7 | gmap brol.grf - /tmp/brol.map
\end{verbatim}

If compressed data handling is enabled, use \texttt{gzip} compressed streams on the fly:

\begin{verbatim}
% echo cmpltw 2 4 7 | gmap brol.grf.gz - /tmp/brol.map.gz
\end{verbatim}

- Map a 32 by 32 bidimensional grid source graph onto a 256-node hypercube, and save the result to file \texttt{/tmp/brol.map}.

\begin{verbatim}
% gmk\_m2 32 32 | gmap - tgt/h8.tgt /tmp/brol.map
\end{verbatim}
• Build the OPEN INVENTOR file `graph.iv` that contains the display of a source graph the source and geometry files of which are named `graph.grf` and `graph.xyz`.

    % gout -Mn -Oi graph.grf graph.xyz - graph.iv

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.

• Given the source and geometry files `graph.grf` and `graph.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 graph.grf tgt/m8x8.tgt | gout graph.grf graph.xyz
    '-Op{c,f,l}' | ghostview -

• Build a 24-node Cube-Connected-Cycles graph target architecture which will be frequently used. Then, map compressed source file `graph.grf.gz` onto it, and save the result to file `/tmp/brol.map`.

    % amk ccc 3 | acpl - /tmp/ccc3.tgt
    % gunzip -c graph.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 `graph.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 graph.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.

10 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.

10.1 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 5.1 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 typedefed types.

10.2 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 multi-level 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 multi-level or Fiduccia-Mattheyses ones.

10.3 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 7.1.5;
- **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 `makegen`.

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.

10.4 Licensing of new methods and of derived works

According to the terms of the CeCILL-C license [6] 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 [47] that had been embedded into SCOTCH 3.3, and gave me 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;
• Alex Pothen kindly gave me a version of his Multiple Minimum Degree algorithm, which was embedded into SCOTCH from version 3.2 to version 3.4;
• Luca Scarano, visiting Erasmus student from the Università degli Studi di Bologna, coded the multi-level graph algorithm in SCOTCH 3.1;
• Yves Secretan contributed to the MinGW32 port;
• David Sherman proofread version 3.2 of this manual.

References


