PT-Scotch and libScotch 5.1
User’s Guide

(version 5.1.11)

François Pellegrini
Bacchus team, INRIA Bordeaux Sud-Ouest
ENSEIRB & 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 PT-Scotch and libScotch, a software package and a software library which compute parallel static mappings and parallel sparse matrix block orderings of graphs. It gives brief descriptions of the algorithms, details the input/output formats, instructions for use, installation procedures, and provides a number of examples.

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

1 Introduction 4
   1.1 Static mapping 4
   1.2 Sparse matrix ordering 5
   1.3 Contents of this document 5

2 The Scotch project 5
   2.1 Description 5
   2.2 Availability 6

3 Algorithms 6
   3.1 Parallel static mapping by Dual Recursive Bipartitioning 6
      3.1.1 Static mapping 6
      3.1.2 Cost function and performance criteria 7
      3.1.3 The Dual Recursive Bipartitioning algorithm 8
      3.1.4 Partial cost function 9
      3.1.5 Parallel graph bipartitioning methods 10
      3.1.6 Mapping onto variable-sized architectures 11
   3.2 Parallel sparse matrix ordering by hybrid incomplete nested dissection 11
      3.2.1 Hybrid incomplete nested dissection 11
      3.2.2 Parallel ordering 12
      3.2.3 Performance criteria 16
   3.3 Changes from version 5.0 16

4 Files and data structures 17
   4.1 Distributed graph files 17

5 Programs 18
   5.1 Invocation 18
   5.2 File names 19
      5.2.1 Sequential and parallel file opening 19
      5.2.2 Using compressed files 20
   5.3 Description 20
      5.3.1 dgmap / dgpart 20
      5.3.2 dgord 22
      5.3.3 dgpart 23
      5.3.4 dgscat 24
      5.3.5 dgtst 24

6 Library 25
   6.1 Running at proper thread level 25
   6.2 Calling the routines of LibScotch 26
      6.2.1 Calling from C 26
      6.2.2 Calling from Fortran 26
      6.2.3 Compiling and linking 27
      6.2.4 Machine word size issues 28
   6.3 Data formats 29
      6.3.1 Distributed graph format 29
      6.3.2 Block ordering format 34
   6.4 Strategy strings 34
      6.4.1 Using default strategy strings 35
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4.2 Parallel mapping strategy strings</td>
<td>36</td>
</tr>
<tr>
<td>6.4.3 Parallel graph bipartitioning strategy strings</td>
<td>37</td>
</tr>
<tr>
<td>6.4.4 Parallel ordering strategy strings</td>
<td>40</td>
</tr>
<tr>
<td>6.4.5 Parallel node separation strategy strings</td>
<td>42</td>
</tr>
<tr>
<td>6.5 Distributed graph handling routines</td>
<td></td>
</tr>
<tr>
<td>6.5.1 SCOTCH_dgraphAlloc</td>
<td>45</td>
</tr>
<tr>
<td>6.5.2 SCOTCH_dgraphInit</td>
<td>45</td>
</tr>
<tr>
<td>6.5.3 SCOTCH_dgraphExit</td>
<td>46</td>
</tr>
<tr>
<td>6.5.4 SCOTCH_dgraphFree</td>
<td>46</td>
</tr>
<tr>
<td>6.5.5 SCOTCH_dgraphLoad</td>
<td>46</td>
</tr>
<tr>
<td>6.5.6 SCOTCH_dgraphSave</td>
<td>47</td>
</tr>
<tr>
<td>6.5.7 SCOTCH_dgraphBuild</td>
<td>48</td>
</tr>
<tr>
<td>6.5.8 SCOTCH_dgraphGather</td>
<td>50</td>
</tr>
<tr>
<td>6.5.9 SCOTCH_dgraphScatter</td>
<td>50</td>
</tr>
<tr>
<td>6.5.10 SCOTCH_dgraphCheck</td>
<td>51</td>
</tr>
<tr>
<td>6.5.11 SCOTCH_dgraphSize</td>
<td>51</td>
</tr>
<tr>
<td>6.5.12 SCOTCH_dgraphData</td>
<td>52</td>
</tr>
<tr>
<td>6.5.13 SCOTCH_dgraphGhst</td>
<td>54</td>
</tr>
<tr>
<td>6.5.14 SCOTCH_dgraphHalo</td>
<td>55</td>
</tr>
<tr>
<td>6.5.15 SCOTCH_dgraphHaloAsync</td>
<td>55</td>
</tr>
<tr>
<td>6.5.16 SCOTCH_dgraphHaloWait</td>
<td>56</td>
</tr>
<tr>
<td>6.6 Distributed graph mapping and partitioning routines</td>
<td></td>
</tr>
<tr>
<td>6.6.1 SCOTCH_dgraphPart</td>
<td>57</td>
</tr>
<tr>
<td>6.6.2 SCOTCH_dgraphMap</td>
<td>58</td>
</tr>
<tr>
<td>6.6.3 SCOTCH_dgraphMapInit</td>
<td>58</td>
</tr>
<tr>
<td>6.6.4 SCOTCH_dgraphMapExit</td>
<td>59</td>
</tr>
<tr>
<td>6.6.5 SCOTCH_dgraphMapSave</td>
<td>60</td>
</tr>
<tr>
<td>6.6.6 SCOTCH_dgraphMapCompute</td>
<td>60</td>
</tr>
<tr>
<td>6.7 Distributed graph ordering routines</td>
<td></td>
</tr>
<tr>
<td>6.7.1 SCOTCH_dgraphOrderInit</td>
<td>61</td>
</tr>
<tr>
<td>6.7.2 SCOTCH_dgraphOrderExit</td>
<td>61</td>
</tr>
<tr>
<td>6.7.3 SCOTCH_dgraphOrderSave</td>
<td>62</td>
</tr>
<tr>
<td>6.7.4 SCOTCH_dgraphOrderSaveMap</td>
<td>62</td>
</tr>
<tr>
<td>6.7.5 SCOTCH_dgraphOrderSaveTree</td>
<td>63</td>
</tr>
<tr>
<td>6.7.6 SCOTCH_dgraphOrderCompute</td>
<td>64</td>
</tr>
<tr>
<td>6.7.7 SCOTCH_dgraphOrderPerm</td>
<td>64</td>
</tr>
<tr>
<td>6.7.8 SCOTCH_dgraphOrderChkDist</td>
<td>65</td>
</tr>
<tr>
<td>6.7.9 SCOTCH_dgraphOrderTreeDist</td>
<td>65</td>
</tr>
<tr>
<td>6.8 Centralized ordering handling routines</td>
<td></td>
</tr>
<tr>
<td>6.8.1 SCOTCH_dgraphCorderInit</td>
<td>66</td>
</tr>
<tr>
<td>6.8.2 SCOTCH_dgraphCorderExit</td>
<td>67</td>
</tr>
<tr>
<td>6.8.3 SCOTCH_dgraphOrderGather</td>
<td>68</td>
</tr>
<tr>
<td>6.9 Strategy handling routines</td>
<td></td>
</tr>
<tr>
<td>6.9.1 SCOTCH_stratInit</td>
<td>68</td>
</tr>
<tr>
<td>6.9.2 SCOTCH_stratExit</td>
<td>69</td>
</tr>
<tr>
<td>6.9.3 SCOTCH_stratSave</td>
<td>69</td>
</tr>
<tr>
<td>6.9.4 SCOTCH_stratDgraphMap</td>
<td>69</td>
</tr>
<tr>
<td>6.9.5 SCOTCH_stratDgraphMapBuild</td>
<td>70</td>
</tr>
<tr>
<td>6.9.6 SCOTCH_stratDgraphOrder</td>
<td>71</td>
</tr>
<tr>
<td>6.9.7 SCOTCH_stratDgraphOrderBuild</td>
<td>71</td>
</tr>
<tr>
<td>6.10 Other data structure routines</td>
<td></td>
</tr>
<tr>
<td></td>
<td>72</td>
</tr>
</tbody>
</table>
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 [10]. Therefore, many studies have been carried out in order to find sub-optimal solutions in reasonable time, including the development of specific algorithms for common topologies such as the hypercube [8, 16]. When the target machine is assumed to have a communication network in the shape of a complete graph, the static mapping problem turns into the partitioning problem, which has also been intensely studied [3, 17, 25, 26, 40]. However, when mapping onto parallel machines the communication network of which is not a bus, not accounting for the topology of the target machine usually leads to worse running times, because simple cut minimization can induce more expensive
long-distance communication [16, 43]; the static mapping problem is gaining pop-
ularity as most of the newer massively parallel machines have a strongly NUMA
architecture.

1.2 Sparse matrix ordering

Many scientific and engineering problems can be modeled by sparse linear systems,
which are solved either by iterative or direct methods. To achieve efficiency with di-
rect 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 num-
bered, and its effects are critical both in terms of memory and of computation costs.

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

1.3 Contents of this document

This document describes the capabilities and operations of PT-SCOTCH, a software
package devoted to parallel static mapping and sparse matrix block ordering. It is
the parallel extension of SCOTCH, a sequential software package devoted to static
mapping, graph and mesh partitioning, and sparse matrix block ordering. While
both packages share a significant amount of code, because PT-SCOTCH transfers
control to the sequential routines of the LIBSCOTCH library when the subgraphs on
which it operates are located on a single processor, the two sets of routines have
a distinct user’s manual. Readers interested in the sequential features of SCOTCH
should refer to the SCOTCH User’s Guide [35].

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

2 The SCOTCH project

2.1 Description

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

It focused first on static mapping, and has resulted in the development of the
Dual Recursive Bipartitioning (or DRB) mapping algorithm and in the study of
several graph bipartitioning heuristics [33], all of which have been implemented in
the SCOTCH software package [37]. Then, it focused on the computation of high-
quality vertex separators for the ordering of sparse matrices by nested dissection, by
extending the work that has been done on graph partitioning in the context of static
mapping [38, 39]. More recently, the ordering capabilities of SCOTCH have been
extended to native mesh structures, thanks to hypergraph partitioning algorithms. New graph partitioning methods have also been recently added [6, 34]. Version 5.0 of Scotch was the first one to comprise parallel graph ordering routines [7], and version 5.1 now offers parallel graph partitioning features, while parallel static mapping will be available in the next release.

2.2 Availability

Starting from version 4.0, which has been developed at INRIA within the 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 [4], which has basically the same features as the GNU LGPL (“Lesser General Public License”) [29]: ability to link the code as a library to any free/libre or even proprietary software, ability to modify the code and to redistribute these modifications. Version 4.0 of Scotch was distributed under the LGPL itself. This version did not comprise any parallel features.

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

3 Algorithms

3.1 Parallel static mapping by Dual Recursive Bipartitioning

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

3.1.1 Static mapping

The parallel program to be mapped onto the target architecture is modeled by a 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 $\tau_{S,T} : V(S) \longrightarrow V(T)$ and $\rho_{S,T} : E(S) \longrightarrow \mathcal{P}(E(T))$, where $\mathcal{P}(E(T))$ denotes the set of all simple loopless paths which can be built from $E(T)$. $\tau_{S,T}(v_S) = v_T$ if process $v_S$ of $S$ is mapped
onto processor $v_T$ of $T$, and $\rho_{S,T}(e_S) = \{e_{S,1}^1, e_{S,2}^1, \ldots, e_{S,n}^1\}$ if communication channel $e_S$ of $S$ is routed through communication links $e_{S,1}^1$, $e_{S,2}^1$, $\ldots$, $e_{S,n}^1$ 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:

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

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

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

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

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

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

and

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

However, since the maximum load imbalance ratio is provided by the user in input of the mapping, the information given by these parameters is of little interest, since what matters is the minimization of the communication cost function under this load balance constraint.

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

$$\mu_{\text{exp}} \stackrel{\text{def}}{=} \frac{\int_C \sum_{e \in E(S)} w_S(e_S)}{\sum_{e \in E(S)} w_S(e_S)}$$

and 

$$\mu_{\text{dil}} \stackrel{\text{def}}{=} \frac{\sum_{e \in E(S)} |\rho_{S,T}(e_S)|}{|E(S)|}.$$ 

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

3.1.3 The Dual Recursive Bipartitioning algorithm

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

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

```
mapping (D, P)  
Set_Of_Processors D;  
Set_Of_Processes P;  
{  
    Set_Of_Processors D0, D1;  
    Set_Of_Processes P0, P1;  
    if (|P| == 0) return; /* If nothing to do. */  
    if (|D| == 1) { /* If one processor in D */  
        result (D, P); /* P is mapped onto it. */  
        return;  
    }  
    (D0, D1) = processor_bipartition (D);  
    (P0, P1) = process_bipartition (P, D0, D1);  
    mapping (D0, P0); /* Perform recursion. */  
    mapping (D1, P1);  
}
```

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

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

- a domain structure, which represents a set of processors in the target architecture;
- a domain bipartitioning function, which, given a domain, bipartitions it into 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 or connected, this distance may be estimated. However, it must respect certain homogeneity properties, such as giving more accurate results as domain sizes decrease. The domain distance function is used by the graph bipartitioning algorithms to compute the communication function to minimize, since it allows the mapper to estimate the dilation of the edges that link vertices which belong to different domains. Using such a distance function amounts to considering
that all routings will use shortest paths on the target architecture, which is how most parallel machines actually do. We have thus chosen that our program would not provide routings for the communication channels, leaving their handling to the communication system of the target machine;

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

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

### 3.1.4 Partial cost function

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

$$f_C'((\tau_S, \rho_S), \sum_{v \in V(S')} \sum_{\{v, v'\} \in E(S')} w_S(\{v, v'\}) |\rho_S(\{v, v'\})|),$$

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

![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.](image)
3.1.5 Parallel graph bipartitioning methods

The core of our parallel recursive mapping algorithm uses process graph parallel bipartitioning methods as black boxes. It allows the mapper to run any type of graph bipartitioning method compatible with our criteria for quality. Bipartitioning jobs maintain an internal image of the current partition, 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 prolongs back the refined partition on the original graph. This method was designed to be able to use expensive partitioning heuristics, such as genetic algorithms, on large graphs, as it dramatically reduces the problem space by several orders of magnitude. However, it was found that, in a multi-level context, it also improves partition quality, by coercing partitions in a problem space that derives from the one which was globally defined at the coarsest level, thus preventing local optimization refinement algorithms to be trapped in local optima of the finer graphs [6].

**Diffusion**

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

**Multi-level**

This algorithm, which has been studied by several authors [3, 18, 25] and should be considered as a strategy rather than as a method since it uses other methods as parameters, repeatedly reduces the size of the graph to bipartition by finding matchings that collapse vertices and edges, computes a partition for the coarsest graph obtained, and prolongs the result back to the original graph, as shown in Figure 2. The multi-level method, when used in conjunction with the banded diffusion method to refine the prolonged partitions at every level, must be used on band graphs only, or on specifically crafted graphs.

---

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

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

usually stabilizes quality irrespective of the number of processors which run the parallel static mapper.

3.1.6 Mapping onto variable-sized architectures

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

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

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

3.2 Parallel sparse matrix ordering by hybrid incomplete nested dissection

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

3.2.1 Hybrid incomplete nested dissection

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

The minimum degree algorithm [42] is a local heuristic that performs its pivot selection by iteratively selecting from the graph a node of minimum degree. It is known to be a very fast and general purpose algorithm, and has received much attention over the last three decades (see for example [1, 13, 31]). However, the algorithm is intrinsically sequential, and very little can be theoretically proved about its efficiency.

The nested dissection algorithm [14] is a global, recursive heuristic algorithm which computes a vertex set $S$ that separates the graph into two parts $A$ and $B$, ordering $S$ with the highest remaining indices. It then proceeds recursively on parts $A$ and $B$ until their sizes become smaller than some threshold value. This ordering guarantees that, at each step, no non-zero term can appear in the factorization process between unknowns of $A$ and unknowns of $B$.

Many theoretical results have been obtained on nested dissection ordering [5, 30], and its divide and conquer nature makes it easily parallelizable. The main issue of the nested dissection ordering algorithm is thus to find small vertex separators that balance the remaining subgraphs as evenly as possible. Provided that good vertex separators are found, the nested dissection algorithm produces orderings which, both in terms of fill-in and operation count, compare favorably [15, 25, 38] to the ones obtained with the minimum degree algorithm [31]. Moreover, the elimination trees induced by nested dissection are broader, shorter, and better balanced, and therefore exhibit much more concurrency in the context of parallel Cholesky factorization [2, 11, 12, 15, 38, 41, and included references].

Due to their complementary nature, several schemes have been proposed to hybridize the two methods [24, 27, 38]. Our implementation is based on a tight coupling of the nested dissection and minimum degree algorithms, that allows each of them to take advantage of the information computed by the other [39].

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

### 3.2.2 Parallel ordering

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

**Nested dissection** As said above, the first level of concurrency relates to the parallelization of the nested dissection method itself, which is straightforward thanks to the intrinsically concurrent nature of the algorithm. Starting from the initial graph, arbitrarily distributed across $p$ processors but preferably balanced in terms of vertices, the algorithm proceeds as illustrated in Figure 3: once a separator has been computed in parallel, by means of a method described below, each of the $p$ processors participates in the building of the distributed induced subgraph corresponding to the first separated part (even if some processors do not have any
Figure 3: Diagram of a nested dissection step for a (sub-)graph distributed across four processors. Once the separator is known, the two induced subgraphs are built and folded (this can be done in parallel for both subgraphs), yielding two subgraphs, each of them distributed across two processors.

vertex of it). This induced subgraph is then folded onto the first ⌈p/2⌉ processors, such that the average number of vertices per processor, which guarantees efficiency as it allows the shadowing of communications by a subsequent amount of computation, remains constant. During the folding process, vertices and adjacency lists owned by the ⌊p/2⌋ sender processors are redistributed to the ⌈p/2⌉ receiver processors so as to evenly balance their loads.

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

At the end of the folding process, every processor has a folded subgraph fragment of one of the two folded subgraphs, and the nested dissection process can recursively proceed independently on each subgroup of p/2 (then p/4, p/8, etc.) processors, until each subgroup is reduced to a single processor. From then on, the nested dissection process will go on sequentially on every processor, using the nested dissection routines of the Scotch library, eventually ending in a coupling with minimum degree methods [39], as described in the previous section.

Graph coarsening The second level of concurrency concerns the computation of separators. The approach we have chosen is the now classical multi-level one [3, 22, 27]. It consists in repeatedly computing a set of increasingly coarser albeit topologically similar versions of the graph to separate, by finding matchings which collapse vertices and edges, until the coarsest graph obtained is no larger than a few hundreds of vertices, then computing a separator on this coarsest graph, and prolonging back this separator, from coarser to finer graphs, up to the original graph. Most often, a local optimization algorithm, such as Kernighan-Lin [28] or Fiduccia-Mattheyses [9] (FM), is used in the uncoarsening phase to refine the partition that is prolonged back at every level, such that the granularity of the solution is the one of the original graph and not the one of the coarsest graph.

The main features of our implementation are outlined in Figure 4. Once the
matching phase is complete, the coarsened subgraph building phase takes place. It can be parametrized so as to allow one to choose between two options. Either all coarsened vertices are kept on their local processors (that is, processors that hold at least one of the ends of the coarsened edges), as shown in the first steps of Figure 4, which decreases the number of vertices owned by every processor and speeds-up future computations, or else coarsened graphs are folded and duplicated, as shown in the next steps of Figure 4, which increases the number of working copies of the graph and can thus reduce communication and increase the final quality of the separators.

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

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

Once all working copies of the coarsened graphs are folded on individual processors, the algorithm enters a multi-sequential phase, illustrated at the bottom of Figure 4: the routines of the sequential Scotch library are used on every processor to complete the coarsening process, compute an initial partition, and prolong it back up to the largest centralized coarsened graph stored on the processor. Then, the partitions are prolonged back in parallel to the finer distributed graphs, selecting the best partition between the two available when prolonging to a level where fold-dup had been performed. This distributed prolongation process is repeated until we obtain a partition of the original graph.

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

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

The computation and use of distributed band graphs is outlined in Figure 5. Given a distributed graph and an initial separator, which can be spread across
Figure 4: Diagram of the parallel computation of the separator of a graph distributed across four processors, by parallel coarsening with folding-with-duplication in the last stages, multi-sequential computation of initial partitions that are locally prolonged back and refined on every processor, and then parallel uncoarsening of the best partition encountered.

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

several processors, vertices that are closer to separator vertices than some small user-defined distance are selected by spreading distance information from all of the separator vertices, using our halo exchange routine. Then, the distributed band graph is created, by adding on every processor two anchor vertices, which are connected to the last layers of vertices of each of the parts. The vertex weight of the anchor vertices is equal to the sum of the vertex weights of all of the vertices they replace, to preserve the balance of the two band parts. Once the separator of the band graph has been refined using some local optimization algorithm, the new separator is prolonged back to the original distributed graph.

Basing on these band graphs, we have implemented a multi-sequential refinement algorithm, outlined in Figure 6. At every distributed uncoarsening step, a distributed band graph is created. Centralized copies of this band graph are then gathered on every participating processor, which serve to run fully independent instances of our sequential FM algorithm. The perturbation of the initial state of the sequential FM algorithm on every processor allows us to explore slightly different solution spaces, and thus to improve refinement quality. Finally, the best refined band separator is prolonged back to the distributed graph, and the uncoarsening process goes on.
Figure 6: Diagram of the multi-sequential refinement of a separator prolonged back from a coarser graph distributed across four processors to its finer distributed graph. Once the distributed band graph is built from the finer graph, a centralized version of it is gathered on every participating processor. A sequential FM optimization can then be run independently on every copy, and the best improved separator is then distributed back to the finer graph.

3.2.3 Performance criteria

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

3.3 Changes from version 5.0

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

A new integer index type has been created in the Fortran interface, to address array indices larger than the maximum value which can be stored in a regular integer. Please refer to Section 7.3 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.

---

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

4 Files and data structures

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

4.1 Distributed graph files

Because even very large graphs are most often stored in the form of centralized files, the distributed graph loading routine of the PT-SCOTCH package, as well as all parallel programs which handle distributed graphs, are able to read centralized graph files in the SCOTCH format and to scatter them on the fly across the available processors (the format of centralized SCOTCH graph files is described in the SCOTCH User’s Guide [35]). However, in order to reduce loading time, a distributed graph format has been designed, so that the different file fragments which comprise distributed graph files can be read in parallel and be stored on local disks on the nodes of a parallel or grid cluster.

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

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

The first line of a distributed graph file holds the distributed graph file version number, which is currently 2. The second line holds the number of files across which the graph data is distributed (referred to as proclnbr in LIBSCOTCH; see for instance Figure 8, page 32, for a detailed example), followed by the number of this file in the sequence (ranging from 0 to (proclnbr - 1), and analogous to proclocnbr in Figure 8). The third line holds the global number of graph vertices (referred to as vertglbnbr), followed by the global number of arcs (inappropriately called edgglbnbr, as it is in fact equal to twice the actual number of edges). The fourth line holds the number of vertices contained in this graph fragment (analogous to vertlocnbr), followed by its local number of arcs (analogous to edgelocnbr). The fifth line holds two figures: the graph base index value (baseval) and a numeric flag.

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

The numeric flag, similar to the one used by the Chaco graph format [19], is made of three decimal digits. A non-zero value in the units indicates that vertex weights are provided. A non-zero value in the tenths indicates that edge weights are provided. A non-zero value in the hundredths indicates that vertex labels are provided; if it is the case, vertices can be stored in any order in the file; else, natural order is assumed, starting from the starting global index of each fragment.

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

Figure 7 shows the contents of two complementary distributed graph files modeling a cube with unity vertex and edge weights and base 0, distributed across two processors.


data in file 1

\begin{verbatim}
2
2  0
8 24
4 12
0  000
3  4  2  1
3  5  3  0
3  6  0  3
3  7  1  2
\end{verbatim}

data in file 2

\begin{verbatim}
2
2  1
8 24
4 12
0  000
3  0  6  5
3  1  7  4
3  2  4  7
3  3  5  6
\end{verbatim}

Figure 7: Two complementary distributed graph files representing a cube distributed across two processors.

\section{Programs}

\subsection{Invocation}

All of the programs comprised in the Scotch and PT-Scotch distributions have been designed to run in command-line mode without any interactive prompting, so that they can be called easily from other programs by means of \texttt{system()} or \texttt{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 \texttt{"-"} 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 \texttt{"-h"} option after the program name. The case of option letters is not significant, except when both the lower and upper cases of a letter have different meanings. When passing parameters to the programs, only the order of file names is significant; options can be put anywhere in the command line, in any order. Examples of use of the different programs of the PT-Scotch project are provided in section 8.
Error messages are standardized, but may not be fully explanatory. However, most of the errors you may run into should be related to file formats, and located in “...Load” routines. In this case, compare your data formats with the definitions given in section 4, and use the dgtst program of the PT-Scotch distribution to check the consistency of your distributed source graphs.

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

5.2 File names

5.2.1 Sequential and parallel file opening

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

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

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

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

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

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

For instance, filename “brol” will lead to the opening of file “brol” on the root processor only, filename “%-brol” (or even “br%-ol”) will lead to the parallel opening of files called “brol” on every processor, and filename “brol%p-%r” will lead to the opening of files “brol2-0” and “brol2-1”, respectively, on each of the two processors on which would run a program of the PT-Scotch distribution.
5.2.2 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 7.

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

5.3 Description

5.3.1 dgmap / dgpart

Synopsis

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

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

Description

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

Since its main purpose is to provide mappings that exhibit high concurrency for communication minimization in the mapped application, it comprises a parallel implementation of the dual recursive bipartitioning algorithm [33], as well as all of the sequential static mapping methods used by its sequential counterpart gmap, to be used on subgraphs located on single processors.

20
**dgpart** is a simplified interface to **dgmap**, which performs graph partitioning instead of static mapping. Consequently, the desired number of parts has to be provided, in lieu of the target architecture.

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

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

**Options**

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

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

- **-c** `flags`
  Tune the default mapping strategy according to the given preference flags. Some of these flags are antagonistic, while others can be combined. See Section 6.4.1 for more information. The currently available flags are the following.
  - `b`  Enforce load balance as much as possible.
  - `q`  Privilege quality over speed. This is the default behavior.
  - `s`  Privilege speed over quality.
  - `t`  Use only safe methods in the strategy.
  - `x`  Favor scalability.

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

- **-h**
  Display the program synopsis.

- **-m** `strat`
  Apply parallel static mapping strategy `strat`. The format of parallel mapping strategies is defined in section 6.4.2. This option is incompatible with options `-b` and `-c`.

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

- **-s** `obj`
  Mask source edge and vertex weights. This option allows the user to “un-weight” weighted source graphs by removing weights from edges and vertices at loading time. `obj` may contain several of the following switches.
  - `e`  Remove edge weights, if any.
  - `v`  Remove vertex weights, if any.

- **-V**
  Print the program version and copyright.
Set verbose mode to `verb`, which may contain several of the following switches.

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

### 5.3.2 dgord

#### Synopsis

dgord [input_graph_file [output_ordering_file [output_log_file]]] options

#### Description

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

Since its main purpose is to provide orderings that exhibit high concurrency for parallel block factorization, it comprises a parallel nested dissection method [14], but sequential classical [31] and state-of-the-art [39] minimum degree algorithms are implemented as well, to be used on subgraphs located on single processors.

Ordering methods can be combined by means of selection, grouping, and condition operators, so as to define ordering strategies, which can be passed to the program by means of the `-o` option. The `-c` option allows the user to set preferences on the behavior of the ordering strategy which is used by default.

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

#### Options

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

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

- `b` Enforce load balance as much as possible.
- `q` Privilege quality over speed. This is the default behavior.
- `s` Privilege speed over quality.
Use only safe methods in the strategy.
Favor scalability.
-h Display the program synopsis.
-m output_mapping_file
Write to output_mapping_file the mapping of graph vertices to column blocks. All of the separators and leaves produced by the nested dissection method are considered as distinct column blocks, which may be in turn split by the ordering methods that are applied to them. Distinct integer numbers are associated with each of the column blocks, such that the number of a block is always greater than the ones of its predecessors in the elimination process, that is, its descendants in the elimination tree. The structure of mapping files is described in detail in the relevant section of the SCOTCH User's Guide [35].
When the geometry of the graph is available, this mapping file may be processed by program gout to display the vertex separators and super-variable amalgamations that have been computed.
-ostrat
Apply parallel ordering strategy strat. The format of parallel ordering strategies is defined in section 6.4.4.
-r num
Set the number of the root process which will be used for centralized file accesses. Set to 0 by default.
-t output_tree_file
Write to output_tree_file the structure of the separator tree. The data that is written resembles much the one of a mapping file: after a first line that contains the number of lines to follow, there are that many lines of mapping pairs, which associate an integer number with every graph vertex index. This integer number is the number of the column block which is the parent of the column block to which the vertex belongs, or −1 if the column block to which the vertex belongs is a root of the separator tree (there can be several roots, if the graph is disconnected).
Combined to the column block mapping data produced by option -m, the tree structure allows one to rebuild the separator tree.
-V Print the program version and copyright.
-verb
Set verbose mode to verb, which may contain several of the following switches.
  a Memory allocation information.
  s Strategy information. This parameter displays the default parallel ordering strategy used by dgord.
  t Timing information.

5.3.3 dgpart

Synopsis

dgpart [number_of_parts [input_graph_file [output_mapping_file [output_log_file]]]] options
Description

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

Save for the `number_of_parts` parameter which replaces the `input_target_file`, the parameters of `dgpart` are identical to the ones of `dgmap`. Please refer to its manual page, in Section 5.3.1, for a description of all of the available options.

5.3.4  `dgscat`

Synopsis

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

Description

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

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

Options

- `-c` Check the consistency of the distributed graph at the end of the graph loading phase.
- `-h` Display the program synopsis.
- `-r num`
  Set the number of the root process which will be used for centralized file accesses. Set to 0 by default.
- `-V` Print the program version and copyright.

5.3.5  `dgtst`

Synopsis

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

Description

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

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

- \( \text{-h} \) Display the program synopsis.
- \( \text{-r num} \)
  
  Set the number of the root process which will be used for centralized file accesses. Set to 0 by default.
- \( \text{-V} \) Print the program version and copyright.

6 Library

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

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

Error handling is performed using the existing sequential routines of the Scotch distribution, which are described in the Scotch User’s Guide [35]. Their use is recalled in Section 6.11.

A ParMeTS compatibility library, called `libptscotchparmetis.a`, is also available. It allows users who were previously using ParMeTS in their software to take advantage of the efficiency of PT-Scotch without having to modify their code. The services provided by this library are described in Section 6.13.

6.1 Running at proper thread level

Since PT-Scotch is based on the MPI API, all processes must call some flavor of `MPI_Init` before using any routine of the library that performs communication. The thread support level of MPI has to be set in accordance to the level required by the library.

If PT-Scotch has been compiled without the `-DSCOTCH_PTHREAD` flag, a call to the simple `MPI_Init` routine will suffice, because no concurrent MPI calls will be performed by library routines. Else, the extended `MPI_Init_thread` initialization routine has to be used, to request the `MPI_THREAD_MULTIPLE` level, and the provided thread support level value returned by the routine must be checked carefully. If your MPI implementation does not provide the `MPI_THREAD_MULTIPLE` level, you will have to recompile PT-Scotch without the `-DSCOTCH_PTHREAD` flag. Else, library calls may cause random bugs in the communication subsystem, resulting in program crashes.
6.2 Calling the routines of \texttt{LIBSCOTCH}

6.2.1 Calling from C

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

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

For instance, the \texttt{SCOTCH\_dgraphInit} and \texttt{SCOTCH\_dgraphLoad} routines, described in section 6.5, can be called from C by using the following code.

```c
#include <stdio.h>
#include <mpi.h>
#include "ptscotch.h"
...
SCOTCH_Dgraph grafdat;
FILE * fileptr;

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

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

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

6.2.2 Calling from Fortran

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

Since all the data structures used in \texttt{LIBSCOTCH} are opaque, equivalent declarations for these structures must be provided in Fortran. These structures must there-
fore be defined as arrays of `DOUBLEPRECISION`, of sizes given in file `ptscotchf.h`, which must be included whenever necessary.

For routines that read or write data using a `FILE *` stream in C, the Fortran counterpart uses an `INTEGER` parameter which is the 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 that are opened using `OPEN` statements, an additional function must be used to obtain the number of the Unix file descriptor from the number of the logical unit. This function is called `PXFFILENO` in the normalized POSIX Fortran API, and files which use it should include the `USE IFPOSIX` directive whenever necessary. An alternate, non normalized, function also exists in most Unix implementations of Fortran, and is called `FNUM`.

For instance, the `SCOTCH_dgraphInit` and `SCOTCH_dgraphLoad` routines, described in sections 6.5.2 and 6.5.5, respectively, can be called from Fortran by using the following code.

```fortran
INCLUDE "ptscotchf.h"
DOUBLEPRECISION GRAFDAT(SCOTCH_DGRAPHDIM)
INTEGER RETVAL

CALL SCOTCHFDGRAPHINIT (GRAFDAT (1), RETVAL)
IF (RETVAL .NE. 0) THEN
...
OPEN (10, FILE='brol.grf')
CALL SCOTCHFDGRAPHLOAD (GRAFDAT (1), FNUM (10), 1, 0, RETVAL)
CLOSE (10)
IF (RETVAL .NE. 0) THEN
...

```

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

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

### 6.2.3 Compiling and linking

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

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

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

If you want to handle errors by yourself, you should not link with library file libptscotcherr.a, but rather provide a SCOTCH_errorPrint() routine. Please refer to Section 6.11 for more information on error handling.

6.2.4 Machine word size issues

Graph indices are represented in Scotch as integer values of type SCOTCH_num. By default, this type equates to the int C type, that is, an integer type of size equal to 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 7.3 for more information on the setting of these compilation flags).

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

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

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

In practice, when Scotch is compiled on a 32-bit architecture so as to use 64-bit SCOTCH_nums, graph indices should be declared as INTEGER*8, while error return values should still be declared as plain INTEGER (that is, INTEGER*4) values. On a 32-bit architecture, irrespective of whether SCOTCH_nums are defined as INTEGER*4 or INTEGER*8 quantities, the SCOTCH_idx type should always be defined as a 64-bit quantity, that is, an INTEGER*8, because it stores differences between memory addresses, which are represented by 64-bit values. The above is no longer a problem if Scotch is compiled such that ints equate 64-bit integers. In this case, there is no need to use any type coercing definition.
Also, the MeIS compatibility library provided by Scotch will not work when SCOTCH_Num are not ints, since the interface of MeIS uses regular ints to represent graph indices. In addition to compile-time warnings, an error message will be issued when one of these routines is called.

6.3 Data formats

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

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

6.3.1 Distributed graph format

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

Distributed source graphs are described by means of adjacency lists. The description of a distributed graph requires several SCOTCH_Num scalars and arrays, as shown for instance in Figures 8 and 9. Some of these data are said to be global, and are duplicated on every process that holds part of the distributed graph; their names contain the “glb” infix. Others are local, that is, their value may differ for each process; their names contain the “loc” or “gst” infix. Global data have the following meaning:

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

- **vertglbnbr**
  Overall number of vertices in the distributed graph.

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

- **procglbnbr**
  Overall number of processes that share distributed graph data.
**proccnttab**
Array holding the current number of local vertices borne by every process.

**procvrttab**
Array holding the global indices from which the vertices of every process are numbered. For optimization purposes, this array has an extra slot which stores a number which must be greater than all of the assigned global indices. For each process $p$, it must be ensured that $procvrttab[p + 1] \geq (procvrttab[p] + proccnttab[p])$, that is, that no process can have more local vertices than allowed by its range of global indices. When the global numbering of vertices is continuous, for each process $p$, $procvrttab[p + 1] = (procvrttab[p] + proccnttab[p])$.

Local data have the following meaning:

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

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

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

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

When all vertex adjacency lists are stored in order in edgeloctab without any empty space between them, it is possible to save memory by not allocating the physical memory for vendloctab. In this case, illustrated in Figure 8, vertloctab is of size vertlocnbr + 1 and vendloctab points to vertloctab + 1. For these graphs, called “compact edge array graphs”, or “compact graphs” for short, vertloctab is sorted in ascending order, vertloctab[baseval] = baseval and vertloctab[baseval + vertlocnbr] = (baseval + edgelocnbr).

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

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

**edgeloctab**
Array, of a size equal at least to $(\max_i(\text{vendloctab}[i]) − baseval)$, holding the adjacency array of every local vertex. For any local vertex $i$, with $baseval \leq i < (baseval + vertlocnbr)$, the global indices of the neighbors of $i$ are stored in edgeloctab from edgeloctab[vertloctab[$i$]] to edgeloctab[vendloctab[$i$] − 1], inclusive.
Since ghost vertices do not have adjacency arrays, because only arcs from local vertices to ghost vertices are recorded and not the opposite, the overall sum of the sizes of all `edgeloctab` arrays is \( e \).

### `edgeloctab`

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

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

Only `edgeloctab` has to be provided by the user. `edgeloctab` is internally computed by PT-Scotch whenever needed, or can be explicitly asked for by the user by calling function `SCOTCH_dgraphGhst`. This array can serve to index user-defined arrays of quantities borne by graph vertices, which can be exchanged between neighboring processes thanks to the `SCOTCH_dgraphHalo` routine documented in Section 6.5.14.

### `edloloctab`

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

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

To add a new vertex, one has to fill `vertloctab[vertnbr - 1]` and `vendloctab[vertnbr - 1]` with the starting and end indices of the adjacency sub-array of the new vertex. Then, the adjacencies of its neighbor vertices must also be updated to account for it. If free space had been reserved at the end of each of the neighbors, one just has to increment the `vendloctab[i]` values of every neighbor `i`, and add the index of the new vertex at the end of the adjacency sub-array. If the sub-array cannot be extended, then it has to be copied elsewhere in the edge array, and both `vertloctab[i]` and `vendloctab[i]` must be updated accordingly. With
Figure 8: Sample distributed graph and its description by LIBSCOTCH arrays using a continuous numbering and compact edge arrays. Numbers within vertices are vertex indices. Top graph is a global view of the distributed graph, labeled with global, continuous, indices. Bottom graphs are local views labeled with local and ghost indices, where ghost vertices are drawn in black. Since the edge array is compact, all vertloctab arrays are of size vertlocnbr + 1, and vendloctab points to vertloctab + 1. edgeloctab edge arrays hold global indices of end vertices, while optional edgegstdtab edge arrays hold global indices of end vertices. edgelocnbr is the local number of arcs (that is, twice the number of edges), including arcs to local vertices as well as to ghost vertices veloloctab and edloloctab are not represented.
<table>
<thead>
<tr>
<th>Duplicated data</th>
<th>baseval</th>
<th>0</th>
<th>vertglbnbr</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>edpglbnbr</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>proccnttab</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>procglbnbr</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>procvrttab</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vendloctab</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>edgeloctab</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>edggsttab</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>vertloctab</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Local data</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>vertlocnbr</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vertgstnbr</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edgaloctnbr</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vertloctab</td>
<td>4 1 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edgaloctab</td>
<td>5 1 2 1 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>edggsttab</td>
<td>6 1 2 1 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vendloctab</td>
<td>7 1 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9: Adjacency structure of the sample graph of Figure 8 with a disjoint edge array and a discontinuous ordering. Both `vertloctab` and `vendloctab` are of size `vertlocnbr`. This allows for the handling of dynamic graphs, the structure of which can evolve with time.
simple housekeeping of free areas of the edge array, dynamic arrays can be updated with as little data movement as possible.

6.3.2 Block ordering format

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

Block orderings are made of the following data:

permtab
Array holding the permutation of the reordered matrix. Thus, if $k = \text{permtab}[i]$, 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 $\text{baseval} \leq c < \text{cblknbr} + \text{baseval}$, contains columns with indices ranging from $\text{rangtab}[i]$ to $\text{rangtab}[i + 1]$, exclusive, in the reordered matrix. Therefore, $\text{rangtab}[\text{baseval}]$ is always equal to $\text{baseval}$, and $\text{rangtab}[\text{cblknbr} + \text{baseval}]$ is always equal to $\text{vertglbnbr} + \text{baseval}$. In order to avoid memory errors when column blocks are all single columns, the size of rangtab must always be one more than the number of columns, that is, $\text{vertglbnbr} + 1$.

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 treetab are based, in the same way as for the other blocking structures. See Figure 10 for a complete example.

6.4 Strategy strings

The behavior of the static mapping and block ordering routines of the libScotch library is parametrized by means of strategy strings, which describe how and when given partitioning or ordering methods should be applied to graphs and subgraphs.
Figure 10: Arrays resulting from the ordering by complete nested dissection of a 4 by 3 grid based from 1. Leftmost grid is the original grid, and rightmost grid is the reordered grid, with separators shown and column block indices written in bold.

6.4.1 Using default strategy strings

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

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

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

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.
Favor scalability as much as possible.

6.4.2 Parallel mapping strategy strings

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

(\texttt{strat})

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

/\texttt{cond}?\texttt{strat1}[:\texttt{strat2}]\

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

\texttt{cond1}|\texttt{cond2}

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

\texttt{cond1}&\texttt{cond2}

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

!\texttt{cond}

Logical not operator. The result of the condition is true only if \texttt{cond} is false.

\texttt{var relop val}

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

\texttt{edge}

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

\texttt{levl}

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

\texttt{load}

The overall sum of the vertex loads of the subgraph. It is equal to \texttt{vert} if the graph has no vertex loads. Integer.

\texttt{mdeg}

The maximum degree of the subgraph. Integer.

\texttt{proc}

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

\texttt{rank}

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

\textit{method[\{parameters\}]}  
Parallel graph mapping method. Available parallel mapping methods are listed below.

The currently available parallel mapping methods are the following.

\textbf{r} Dual recursive bipartitioning method. The parameters of the dual recursive bipartitioning method are given below.

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

\textbf{sep=strat}  
Set the parallel graph bipartitioning strategy that is used on every current job of the recursion tree. Parallel graph bipartitioning strategies are described below, in section 6.4.3.

\section{6.4.3 Parallel graph bipartitioning strategy strings}
A parallel graph bipartitioning strategy is made of one or several parallel graph bipartitioning methods, which can be combined by means of strategy operators. Strategy operators are listed below, by increasing precedence.

\textbf{strat1 | strat2}  
Selection operator. The result of the selection is the best bipartition of the two that are obtained by the distinct application of \textit{strat1} and \textit{strat2} to the current bipartition.

\textbf{strat1 strat2}  
Combination operator. Strategy \textit{strat2} is applied to the bipartition resulting from the application of strategy \textit{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.

\textbf{(strat)}  
Grouping operator. The strategy enclosed within the parentheses is treated as a single bipartitioning method.

\textbf{/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.

\textbf{cond1 | cond2}  
Logical or operator. The result of the condition is true if \textit{cond1} or \textit{cond2} are true, or both.
**cond1**&**cond2**
Logical and operator. The result of the condition is true only if both **cond1** and **cond2** are true.

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

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

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

**levl**
The level of the subgraph in the bipartition or multi-level tree, starting from zero for the initial graph at the root of the tree. Integer.

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

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

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

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

**vert**
The number of vertices of the current subgraph. Integer.

The currently available parallel vertex separation methods are the following.

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

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

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

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

Parallel diffusion method. This method, presented in its sequential formulation in [34], flows two kinds of antagonistic liquids, scotch and anti-scotch, from two source vertices, and sets the new frontier as the limit between vertices which contain scotch and the ones which contain anti-scotch. Because selecting the source vertices is essential to the obtaining 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.

\[ \text{dif} = \text{rat} \]
Fraction of liquid which is diffused to neighbor vertices at each pass. To achieve convergence, the sum of the \( \text{dif} \) and \( \text{rem} \) 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.

\[ \text{pass} = \text{nbr} \]
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 \( \text{dif} \) and \( \text{rem} \) coefficients.

\[ \text{rem} = \text{rat} \]
Fraction of liquid which remains on vertices at each pass. See above.

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

\[ \text{asc} = \text{strat} \]
Set the strategy that is used to refine the distributed bipartition obtained at ascending levels of the uncoarsening phase by prolongation of the bipartition computed for coarser graphs. This strategy is not applied to the coarsest graph, for which only the \text{low} strategy is used.

\[ \text{dlevl} = \text{nbr} \]
Set the minimum level after which duplication is allowed in the folding process. A value of \(-1\) results in duplication being always performed when folding.

\[ \text{dvert} = \text{nbr} \]
Set the average number of vertices per process under which the folding process is performed during the coarsening phase.

\[ \text{low} = \text{strat} \]
Set the strategy that is used to compute the bipartition of the coarsest distributed graph, at the lowest level of the coarsening process.

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

\texttt{vert=nbr}

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

\texttt{q}

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

\texttt{strat=}\texttt{strat}

Set the sequential edge separation strategy that is used to refine the bipartition of the centralized graph. For a description of all of the available sequential bipartitioning methods, please refer to the \textit{Scotch User’s Guide} [35].

\texttt{x}

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

\texttt{sbbt=nbr}

Number of sub-buckets to sort communication gains. 5 is a common value.

\texttt{z}

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

### 6.4.4 Parallel ordering strategy strings

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

\texttt{(strat)}

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

\texttt{/cond?strat1::strat2;}  

Condition operator. According to the result of the evaluation of condition \texttt{cond}, either \texttt{strat1} or \texttt{strat2} (if it is present) is applied. The condition applies to the characteristics of the current node of the separators tree, and can be built from logical and relational operators. Conditional operators are listed below, by increasing precedence.
cond1 | cond2
Logical or operator. The result of the condition is true if cond1 or cond2
are true, or both.

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

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

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

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

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

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

mdeg
The maximum degree of the subgraph. Integer.

proc
The number of processes on which the current subgraph is dis-
tributed at this level of the separators tree. Integer.

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

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

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

The currently available parallel ordering methods are the following.

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

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

ose=strat
Set the parallel ordering strategy that is used on every distributed separ-
arator of the separators tree.
Set the sequential ordering strategy that is used on every centralized subgraph of the separators tree, once the nested dissection process has gone far enough such that the number of processes handling some subgraph is restricted to one.

Set the parallel node separation strategy that is used on every current leaf of the separators tree to make it grow. Parallel node separation strategies are described below, in section 6.4.5.

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

Set the sequential ordering strategy that is applied to the centralized graph. For a description of all of the available sequential ordering methods, please refer to the Scotch User’s Guide [35].

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

### 6.4.5 Parallel node separation strategy strings

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

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.

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.

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

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.

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

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

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

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

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

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

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

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

**vert**
The number of vertices of the current subgraph. Integer.

The currently available parallel vertex separation methods are the following.

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

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

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

**m** Parallel vertex multi-level method. The parameters of the vertex multi-level method are listed below.
asc=$strat$
Set the strategy that is used to refine the distributed vertex separators obtained at ascending levels of the uncoarsening phase by prolongation of the separators computed for coarser graphs. This strategy is not applied to the coarsest graph, for which only the low strategy is used.

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

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

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

rat=$rat$
Set the threshold maximum coarsening ratio over which graphs are no longer coarsened. The ratio of any given coarsening cannot be less 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 node vertices than the minimum number of vertices allowed.

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

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

strat=$strat$
Set the sequential vertex separation strategy that is used to refine the separator of the centralized graph. For a description of all of the available sequential methods, please refer to the Scotch User’s Guide [35].

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

6.5.1 SCOTCH_dgraphAlloc

Synopsis

SCOTCH_Dgraph * SCOTCH_dgraphAlloc (void)

Description

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

Return values

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

6.5.2 SCOTCH_dgraphInit

Synopsis

int SCOTCH_dgraphInit (SCOTCH_Dgraph * grafptr,
MPI_Comm comm)

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

Description

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

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

When the distributed graph is no longer of use, call function SCOTCH_dgraph
Exit to free its internal communication structures.
Return values

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

\section*{6.5.3 \texttt{SCOTCH\_dgraphExit}}

Synopsis

\begin{verbatim}
void SCOTCH\_dgraphExit (SCOTCH\_Dgraph * grafptr)
scotchfdgraphexit (doubleprecision (*) grafdat)
\end{verbatim}

Description

The \texttt{SCOTCH\_dgraphExit} function frees the contents of a \texttt{SCOTCH\_Dgraph} structure previously initialized by \texttt{SCOTCH\_dgraphInit}. All subsequent calls to \texttt{SCOTCH\_dgraph} routines other than \texttt{SCOTCH\_dgraphInit}, using this structure as parameter, may yield unpredictable results.

\section*{6.5.4 \texttt{SCOTCH\_dgraphFree}}

Synopsis

\begin{verbatim}
void SCOTCH\_dgraphFree (SCOTCH\_Dgraph * grafptr)
scotchfdgraphfree (doubleprecision (*) grafdat)
\end{verbatim}

Description

The \texttt{SCOTCH\_dgraphFree} function frees the graph data of a \texttt{SCOTCH\_Dgraph} structure previously initialized by \texttt{SCOTCH\_dgraphInit}, but preserves its internal communication data structures. This call is equivalent to a call to \texttt{SCOTCH\_dgraphExit} immediately followed by a call to \texttt{SCOTCH\_dgraphInit} with the same communicator as in the previous \texttt{SCOTCH\_dgraphInit} call. Consequently, the given \texttt{SCOTCH\_Dgraph} structure remains ready for subsequent calls to any distributed graph handling routine of the LIBSCOTCH library.

\section*{6.5.5 \texttt{SCOTCH\_dgraphLoad}}

Synopsis

\begin{verbatim}
int SCOTCH\_dgraphLoad (SCOTCH\_Dgraph * grafptr,
FILE * stream,
SCOTCH\_Num baseval,
SCOTCH\_Num flagval)
\end{verbatim}
The `SCOTCH_dgraphLoad` routine fills the `SCOTCH_Dgraph` structure pointed to by `grafptr` with the centralized or distributed source graph description available from one or several streams `stream` in the `SCOTCH` graph formats (please refer to section 4.1 for a description of the distributed graph format, and to the `SCOTCH User’s Guide` [35] for the centralized graph format).

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

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

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

- 0: Keep vertex and edge weights if they are present in the `stream` data.
- 1: Remove vertex weights. The graph read will have all of its vertex weights set to one, regardless of what is specified in the `stream` data.
- 2: Remove edge weights. The graph read will have all of its edge weights set to one, regardless of what is specified in the `stream` data.

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

Return values

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

### 6.5.6 `SCOTCH_dgraphSave`

**Synopsis**

```c
int SCOTCH_dgraphSave (const SCOTCH_Dgraph * grafptr,
                        FILE * stream)
```
scotchfdgraphsave (doubleprecision (*) grafdat,
        integer   fildes,
        integer   ierr)

Description

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

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

Return values

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

6.5.7 SCOTCH_dgraphBuild

Synopsis

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

```c
scotchfdgraphbuild (doubleprecision (*) grafdat,
        integer* num   baseval,
        integer* num   vertlocnbr,
        integer* num   vertlocmax,
        integer* num   (*) vertloctab,
        integer* num   (*) vendloctab,
        integer* num   (*) veloloctab,
        integer* num   (*) vlblocltab,
        integer* num   edgelocnbr,
        integer* num   edgelocsiz,
        integer* num   (*) edgeloctab,
        integer* num   (*) edgegsttab,
        integer* num   (*) edloloctab,
        integer        ierr)
```
The `SCOTCH_dgraphBuild` routine fills the distributed source graph structure pointed to by `grafptr` with all of the data that are passed to it.

`baseval` is the graph base value for index arrays (typically 0 for structures built from C and 1 for structures built from Fortran). `vertlocnbr` is the number of local vertices on the calling process, used to create the `proccnttab` array. `vertlocmax` is the maximum number of local vertices to be created on the calling process, used to create the `procvrttab` array of global indices, and which must be set to `vertlocnbr` for graphs without holes in their global numbering. `vertloctab` is the local adjacency index array, of size `(vertlocnbr+1)` if the edge array is compact (that is, if `vendloctab` equals `vertloctab` + 1 or NULL), or of size `vertlocnbr` else. `vendloctab` is the adjacency end index array, of size `vertlocnbr` if it is disjoint from `vertloctab`. `veloctab` is the local vertex load array, of size `vertlocnbr` if it exists. `vblloctab` is the local vertex label array, of size `vertlocnbr` if it exists. `edgelocsiz` is lower-bounded by the minimum size of the edge array required to encompass all used adjacency values; it is therefore at least equal to the maximum of the `vendloctab` entries, over all local vertices, minus `baseval`; it can be set to `edgelocnbr` if the edge array is compact. `edgeloctab` is the local adjacency array, of size at least `edgelocsiz`, which stores the global indices of end vertices. `edgegsttab` is the adjacency array, of size at least `edgelocsiz`, if it exists; if `edgegsttab` is given, it is assumed to be a pointer to an empty array to be filled with ghost vertex data computed by `SCOTCH_dgraphGhst` whenever needed by communication routines such as `SCOTCH_dgraphHalo`. `edloloctab` is the arc load array, of size `edgelocsiz` if it exists.

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

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

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

Return values

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

6.5.8 SCOTCH_dgraphGather

Synopsis

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

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

Description

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

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

Return values

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

6.5.9 SCOTCH_dgraphScatter

Synopsis

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

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

Description

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

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

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

Return values

**SCOTCH\(_{\text{dgraph}}\)Scatter** returns 0 if the graph structure has been successfully scattered, and 1 else.

### 6.5.10 SCOTCH\(_{\text{dgraphCheck}}\)

**Synopsis**

```c
int SCOTCH\(_{\text{dgraphCheck}}\) (const SCOTCH\(_{\text{Dgraph}}\) * grafptr)
```

**Description**

The **SCOTCH\(_{\text{dgraphCheck}}\)** routine checks the consistency of the given **SCOTCH\(_{\text{Dgraph}}\)** structure. It can be used in client applications to determine if a graph which has been created from user-generated data by means of the **SCOTCH\(_{\text{dgraphBuild}}\)** routine is consistent, prior to calling any other routines of the **LIBSCOTCH** library which would otherwise return internal error messages or crash the program.

Return values

**SCOTCH\(_{\text{dgraphCheck}}\)** returns 0 if graph data are consistent, and 1 else.

### 6.5.11 SCOTCH\(_{\text{dgraphSize}}\)

**Synopsis**

```c
void SCOTCH\(_{\text{dgraphSize}}\) (const SCOTCH\(_{\text{Dgraph}}\) * grafptr,
    SCOTCH\(_{\text{Num}}\) * vertglbptr,
    SCOTCH\(_{\text{Num}}\) * vertlocptr,
    SCOTCH\(_{\text{Num}}\) * edgelbptr,
    SCOTCH\(_{\text{Num}}\) * edgelocptr)
```

**Description**

The **SCOTCH\(_{\text{dgraphSize}}\)** routine returns the number of vertices and edges in the distributed graph structure. It can be used to determine the size of the graph prior to performing any operations that require knowledge of the graph size.
**Description**

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

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

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

### 6.5.12 SCOTCH_dgraphData

**Synopsis**

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

52
scotchfdgraphdata (doubleprecision (*) grafdat,
integer*num (*) indxtab,
integer*num baseval,
integer*num vertglbptr,
integer*num vertlocnbr,
integer*num vertlocmax,
integer*num vertgstarb,
integer*idx vertlocidx,
integer*idx vendlocidx,
integer*idx velocidx,
integer*idx vlbllocidx,
integer*idx edgelocptr,
integer*idx edgelocsz,
integer*idx edgelocidx,
integer*idx edggestidx,
integer*idx edgloctab,
integer comm)

Description

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

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

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

Since there are no pointers in Fortran, a specific mechanism is used to allow users to access graph arrays. The scotchfdgraphdata routine is passed an integer array, the first element of which is used as a base address from which all other array indices are computed. Therefore, instead of returning references, the routine returns integers, which represent the starting index of each of the relevant arrays with respect to the base input array, or vertlocidx, the index of vertloctab, if they do not exist. For instance, if some base array myarray (1) is passed as parameter indxtab, then the first cell of array vertloctab will be accessible as myarray(vertlocidx). In order for this feature to behave properly, the indxtab array must be word-aligned with the graph arrays. This is automatically enforced on most systems, but some care should be taken on systems that allow to access data that is not word-aligned. On such systems, declaring the array after a dummy doubleprecision array can coerce the compiler into enforcing the proper alignment. The integer value returned in comm is the communicator itself, not its index with respect to indxtab. Also, on 32/64 architectures, such indices can be larger than the size of a regular INTEGER. This is why the indices to be returned are defined by means of a specific integer type. See Section 6.2.4 for more information on this issue.

6.5.13 SCOTCH_dgraphGhst

Synopsis

```c
int SCOTCH_dgraphGhst (SCOTCH_Dgraph * const grafptr)
scotchfdgraphghst (doubleprecision (*) grafdat,
        integer                   ierr)
```

Description

The SCOTCH_dgraphGhst routine fills the edgegsttab arrays of the distributed graph structure pointed to by grafptr with the local and ghost vertex indices corresponding to the global vertex indices contained in its edgeloctab arrays, according to the semantics described in Section 6.3.1.

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

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

6.5.14 SCOTCH_dgraphHalo

Synopsis

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

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

Description

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

When the MPI data type to be used is not a collection of contiguous entries, great care should be taken in the definition of the upper bound of the type (by using the `MPI_INDEX` pseudo-datatype), such that when asking MPI to send a certain number of elements of the said type located at some address, contiguous areas in memory will be considered. Please refer to the MPI documentation regarding the creation of derived datatypes [32, Section 3.12.3] for more information.

To perform its data exchanges, the SCOTCH_dgraphHalo routine requires ghost vertex management data provided by the SCOTCH_dgraphGhst routine. Therefore, the `edggesttab` array returned by the SCOTCH_dgraphData routine will always be valid after a call to SCOTCH_dgraphHalo, if it was not already.

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

Return values

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

6.5.15 SCOTCH_dgraphHaloAsync

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

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

Description

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

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

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

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

Return values

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

6.5.16 SCOTCH_dgraphHaloWait

Synopsis
int SCOTCH_dgraphHaloWait (SCOTCH_DgraphHaloReq * const requptr)

scotchfdgraphhalowait (doubleprecision (*) requptr,
                     integer           ierr)

Description

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

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

Return values

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

6.6 Distributed 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.

6.6.1 SCOTCH_dgraphPart

Synopsis

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

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

Description

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

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

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

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

6.6.2 SCOTCH_dgraphMap

Synopsis

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

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

Description

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

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

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

Attention: version 5.1 of Scotch does not allow yet to map distributed graphs onto target architectures which are not complete graphs. This restriction will be removed in the next release.

Return values

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

6.6.3 SCOTCH_dgraphMapInit

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

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

Description

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

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

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

Return values

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

6.6.4 SCOTCH_dgraphMapExit

Synopsis

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

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

Description

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

Synopsis

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

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

Description

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

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

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

Return values

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

6.6.6 SCOTCH_dgraphMapCompute

Synopsis

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

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

Description

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

On return, every cell of the distributed mapping array (see section 6.6.3) holds the number of the target vertex to which the corresponding source vertex is
mapped. The numbering of target values is \textit{not} based: target vertices are numbered from 0 to the number of target vertices, minus 1.

\textbf{Attention:} version 5.1 of \textsc{scotch} does not allow yet to map distributed graphs onto target architectures which are not complete graphs. This restriction will be removed in the next release.

\textbf{Return values}

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

\section*{6.7 Distributed graph ordering routines}

\subsection*{6.7.1 \textsc{scotch\_dgraphOrderInit}}

\textbf{Synopsis}

\begin{verbatim}
int SCOTCH_dgraphOrderInit (const SCOTCH_Dgraph * grapfptr,
                          SCOTCH_Dordering * ordeptr)
scotchfdgraphorderinit (doubleprecision (*) grafdat,
                       doubleprecision (*) ordedat,
                       integer ierr)
\end{verbatim}

\textbf{Description}

The \textsc{scotch\_dgraphOrderInit} routine initializes the distributed ordering structure pointed to by \texttt{ordeptr} so that it can be used to store the results of the parallel ordering of the associated distributed graph, to be computed by means of the \textsc{scotch\_dgraphOrderCompute} routine.

The \textsc{scotch\_dgraphOrderInit} routine should be the first function to be called upon a \textsc{scotch\_Dordering} structure for ordering distributed graphs. When the ordering structure is no longer of use, the \textsc{scotch\_dgraphOrderExit} function must be called, in order to free its internal structures.

\textbf{Return values}

\textsc{scotch\_dgraphOrderInit} returns 0 if the distributed ordering structure has been successfully initialized, and 1 else.

\subsection*{6.7.2 \textsc{scotch\_dgraphOrderExit}}

\textbf{Synopsis}

\begin{verbatim}
void SCOTCH_dgraphOrderExit (const SCOTCH_Dgraph * grapfptr,
                            SCOTCH_Dordering * ordeptr)
scotchfgraphdorderexit (doubleprecision (*) grafdat,
                       doubleprecision (*) ordedat)
\end{verbatim}
Description

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

6.7.3 SCOTCH_dgraphOrderSave

Synopsis

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

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

Description

The SCOTCH_dgraphOrderSave routine saves the contents of the SCOTCH_Dordering structure pointed to by ordeptr to stream stream, in the SCOTCH ordering format. Please refer to the SCOTCH User’s Guide [35] for more information about this format.

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

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

Return values

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

6.7.4 SCOTCH_dgraphOrderSaveMap

Synopsis

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

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

62
The SCOTCH_dgraphOrderSaveMap routine saves the block partitioning data associated with the SCOTCH_Dordering structure pointed to by ordeptr to stream stream, in the SCOTCH mapping format. A target domain number is associated with every block, such that all node vertices belonging to the same block are shown as belonging to the same target vertex. The resulting mapping file can be used by the gout program to produce pictures showing the different separators and blocks. Please refer to the SCOTCH User’s Guide for more information on the SCOTCH mapping format and on gout.

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

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

Return values

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

6.7.5 SCOTCH_dgraphOrderSaveTree

Synopsis

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

scotchfdgraphordersavetree (doubleprecision (*) grafdat,  
doubleprecision (*) ordedat,  
integer fildes,  
inerr)

Description

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

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

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

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

Return values

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

6.7.6 SCOTCH_dgraphOrderCompute

Synopsis

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

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

Description

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

Return values

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

6.7.7 SCOTCH_dgraphOrderPerm

Synopsis

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

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

Description

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

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

6.7.8 SCOTCH_dgraphOrderCblkDist

Synopsis

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

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

Description

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

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

Return values

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

6.7.9 SCOTCH_dgraphOrderTreeDist

Synopsis

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

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

Description

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

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

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

Each of the \( \text{treeglbtab} \) and \( \text{sizeglbtab} \) arrays must be large enough to hold a number of \( \text{SCOTCH}\_\text{Num} \)s equal to the number of distributed elimination tree nodes and column blocks, as returned by the \( \text{SCOTCH\_dgraphOrderCblkDist} \) routine.

Return values

\( \text{SCOTCH\_dgraphOrderTreeDist} \) returns 0 if the arrays describing the distributed part of the distributed tree structure have been successfully filled, and 1 else.

### 6.8 Centralized ordering handling routines

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

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

#### 6.8.1 \( \text{SCOTCH\_dgraphCorderInit} \)

Synopsis

```c
int SCOTCH\_dgraphCorderInit (const SCOTCH\_Dgraph * grafptr, SCOTCH\_Ordering * cordptr, SCOTCH\_Num * permtab, SCOTCH\_Num * peritab, SCOTCH\_Num * cblkptr, SCOTCH\_Num * rangtab, SCOTCH\_Num * treetab)
```
The *SCOTCH_dgraphCorderInit* routine fills the centralized ordering structure pointed to by *cordptr* with all of the data that are passed to it. This routine is the equivalent of the *SCOTCH_graphOrderInit* routine of the *Scotch* sequential library, except that it takes a distributed graph as input. It is used to initialize a centralized ordering structure on which a distributed ordering will be centralized by means of the *SCOTCH_dgraphOrderGather* routine. Only the process on which distributed ordering data is to be centralized has to handle a centralized ordering structure.

*permtab* is the ordering permutation array, of size *vertglbnbr*, *peritab* is the inverse ordering permutation array, of size *vertglbnbr*, *cblkptr* is the pointer to a *SCOTCH_Num* that will receive the number of produced column blocks, *rangtab* is the array that holds the column block span information, of size *vertglbnbr* + 1, and *treetab* is the array holding the structure of the separators tree, of size *vertglbnbr*. Please refer to Section 6.3.2 for an explanation of their semantics. Any of these five output fields can be set to NULL if the corresponding information is not needed. Since, in Fortran, there is no null reference, passing a reference to *grafptr* will have the same effect.

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

**Return values**

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

### 6.8.2 SCOTCH_dgraphCorderExit

**Synopsis**

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

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

**Description**
6.8.3 SCOTCH_dgraphOrderGather

Synopsis

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

Description

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

Return values

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

6.9 Strategy handling routines

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

6.9.1 SCOTCH_stratInit

Synopsis

```
int SCOTCH_stratInit (SCOTCH_Strat * straptr)
```

Description

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

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

6.9.2 SCOTCH_stratExit

Synopsis

void SCOTCH_stratExit (SCOTCH_Strat * archptr)
scotchfstratexit (doubleprecision (*) stradat)

Description

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

6.9.3 SCOTCH_stratSave

Synopsis

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

Description

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

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

Return values

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

6.9.4 SCOTCH_stratDgraphMap

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

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

Description

The SCOTCH_stratDgraphMap routine fills the strategy structure pointed to by straptr with the distributed graph mapping strategy string pointed to by string. The format of this strategy string is described in Section 6.4.2. From this point, strategy strat can only be used as a distributed graph mapping strategy, to be used by functions SCOTCH_dgraphPart, SCOTCH_dgraphMap or SCOTCH_dgraphMapCompute. This routine must be called on every process with the same strategy string.

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

Return values

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

6.9.5 SCOTCH_stratDgraphMapBuild

Synopsis

int SCOTCH_stratDgraphMapBuild (SCOTCH_Strat * straptr,  
const SCOTCH_Num flagval,  
const SCOTCH_Num procnbr,  
const SCOTCH_Num part nbr,  
const double balrat)

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

Description

The SCOTCH_stratDgraphMapBuild routine fills the strategy structure pointed to by straptr with a default mapping strategy tuned according to the preference flags passed as flagval and to the desired number of parts partnbr and imbalance ratio balrat, to be used on procnbr processes. From this point, the strategy structure can only be used as a parallel mapping strategy, to be used by function SCOTCH_dgraphMap, for instance. See Section 6.4.1 for a description of the available flags.
Return values

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

6.9.6 `SCOTCH_stratDgraphOrder`

Synopsis

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

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

Description

The `SCOTCH_stratDgraphOrder` routine fills the strategy structure pointed to by `straptr` with the distributed graph ordering strategy string pointed to by `string`. The format of this strategy string is described in Section 6.4.4. From this point, strategy `strat` can only be used as a distributed graph ordering strategy, to be used by function `SCOTCH_dgraphOrderCompute`. This routine must be called on every process with the same strategy string.

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

Return values

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

6.9.7 `SCOTCH_stratDgraphOrderBuild`

Synopsis

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

```c
scotchfstratdgraphorderbuild (doubleprecision (*) stradat,
integer* num flagval,
integer* num partnbr,
doubleprecision balrat,
integer ierr)
```

Description

The `SCOTCH_stratDgraphOrderBuild` 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 number of parts `partnbr`
and imbalance ratio balrat, to be used on procnbr processes. From this point, the strategy structure can only be used as a parallel ordering strategy, to be used by function SCOTCH_dgraphOrder, for instance. See Section 6.4.1 for a description of the available flags.

Return values

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

6.10 Other data structure routines

6.10.1 SCOTCH_dmapAlloc

Synopsis

SCOTCH_Dmapping * SCOTCH_dmapAlloc (void)

Description

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

Return values

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

6.10.2 SCOTCH_dorderAlloc

Synopsis

SCOTCH_Dordering * SCOTCH_dorderAlloc (void)

Description

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

Return values

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

6.11 Error handling routines

The handling of errors that occur within library routines is often difficult, because library routines should be able to issue error messages that help the application programmer to find the error, while being compatible with the way the application handles its own errors.
To match these two requirements, all the error and warning messages produced by the routines of the LIBSCOTCH library are issued using the user-definable variable-length argument routines SCOTCH_errorPrint and SCOTCH_errorPrintW. Thus, one can redirect these error messages to his own error handling routines, and can choose if he wants his program to terminate on error or to resume execution after the erroneous function has returned.

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

6.11.1 SCOTCH_errorPrint

Synopsis

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

Description

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

6.11.2 SCOTCH_errorPrintW

Synopsis

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

Description

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

6.11.3 SCOTCH_errorProg

Synopsis

    void SCOTCH_errorProg (const char * progstr)

Description

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

6.12.1 SCOTCH_randomReset

Synopsis

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

Description

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

6.13 ParMETIS compatibility library

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

6.13.1 ParMETIS_V3_NodeND

Synopsis

```c
void ParMETIS_V3_NodeND (const int * const vtxdist,
                         const int * const xadj,
                         const int * const adjncy,
                         const int * const numflag,
                         const int * const options,
                         int * const order,
                         int * const sizes,
                         MPI_Comm * comm)
```
parmetis\_v3\_nodend (integer (*) vtdist, 
   integer (*) xadj, 
   integer (*) adjncy, 
   integer numflag, 
   integer (*) options, 
   integer (*) order, 
   integer (*) sizes, 
   integer comm)

Description

The ParMETIS\_v3\_NodeND function performs a nested dissection ordering of the distributed graph passed as arrays vtdist, xadj and adjncy, using the default PT-SCOTCH ordering strategy. Unlike for ParMETIS, this routine will compute an ordering even when the number of processors on which it is run is not a power of two. The options array is not used. When the number of processors is a power of two, the contents of the sizes array is equivalent to the one returned by the original ParMETIS\_v3\_NodeND routine, else it is filled with $-1$ values.

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

Similarly, as there is no ParMETIS\_v3\_NodeWND routine in ParMETIS, users willing to order distributed graphs with node weights should directly call the PT-SCOTCH routines.

6.13.2 ParMETIS\_v3\_PartGeomKway

Synopsis

void ParMETIS\_v3\_PartGeomKway (const int * const vtdist, 
   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 ndims, 
   const float * const xyz, 
   const int * const ncon, 
   const int * const nparts, 
   const float * const tpwgts, 
   const float * const ubvec, 
   const int * const options, 
   int * const edgecut, 
   int * const part, 
   MPI\_Comm * comm)

75
parmetis_v3_partgeomkway (integer (*) vtxdist,
    integer (*) xadj,
    integer (*) adjncy,
    integer (*) vwgt,
    integer (*) adjwgt,
    integer wgtflag,
    integer numflag,
    integer ndims,
    float (*) xyz,
    integer ncon,
    integer nparts,
    float (*) tpwgts,
    float (*) ubvec,
    integer (*) options,
    integer edgecut,
    integer (*) part,
    integer comm)

Description

The ParMETIS V3 PartGeomKway function computes a partition into nparts parts of the distributed graph passed as arrays vtxdist, xadj and adjncy, using the default PT-SCOTCH mapping strategy. The partition is returned in the form of the distributed vector part, which holds the indices of the parts to which every vertex belongs, from 0 to (nparts − 1).

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

6.13.3 ParMETIS_V3_PartKway

Synopsis

void ParMETIS_V3_PartKway (const int * const vtxdist,
    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 ncon,
    const int * const nparts,
    const float * const tpwgts,
    const float * const ubvec,
    const int * const options,
    int * const edgecut,
    int * const part,
    MPI_Comm * comm)
parmetis_v3_partkway (integer (*) vtxdist,  
    integer (*) xadj,  
    integer (*) adjncy,  
    integer (*) vwgt,  
    integer (*) adjwgt,  
    integer wgtflag,  
    integer numflag,  
    integer ncon,  
    integer nparts,  
    float (*) tpwgts,  
    float (*) ubvec,  
    integer (*) options,  
    integer edgecut,  
    integer (*) part,  
    integer comm)

Description

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

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

7 Installation

Version 5.1 of the SCOTCH software package, which contains the PT-SCOTCH routines, is distributed as free/libre software under the CeCILL-C free/libre software license [4], which is very similar to the GNU LGPL license. Therefore, it is not distributed as a set of binaries, but instead in the form of a source distribution, which can be downloaded from the SCOTCH web page at http://www.labri.fr/~pelegrin/scotch/.

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

The extraction process will create a scotch_5.1.11 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.

7.1 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. If the flag is defined, make sure to use the `MPI_Init_thread` MPI routine to initialize the communication subsystem, at the `MPI_THREAD_MULTIPLE` level (see Section 6.1).

### 7.2 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 `libbz2-1-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.

### 7.3 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 MeRIIS compatibility library will not be usable. It is usually safer and cleaner to tune your C and Fortran compilers to make them interpret `int` and `INTEGER` types as 32 or 64 bit values, than to use the aforementioned flags and coerce type lengths in your own code.

Fortran users also have to take care of another size issue: since there are no pointers in Fortran 77, the Fortran interface of some routines converts pointers to be returned into integer indices with respect to a given array (e.g. see Section 6.5.12). For 32/64 architectures, such indices can be larger than the size of a regular `INTEGER`. This is why the indices to be returned are defined by means of a specific integer type, `SCOTCH_IIdx`. To coerce the length of this index type to 32 or 64 bits, one can use the “-DIDXSIZE32” or “-DIDXSIZE64” flags, respectively, or else use the “-DIDX=” definition, at compile time. For instance, adding “-DIDX=long long” to the `CFLAGS` variable in the `Makefile.inc` file to be placed at the root of the source tree will equate all `SCOTCH_IIdx` integers to C `long long` integers. By default, when the size of `SCOTCH_IIdx` is not explicitly defined, it is assumed to be the same as the size of `SCOTCH_Num`. 

78
8 Examples

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

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

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

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

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

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

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

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

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

- Recompile a program that used PARMeNS so that it uses PT-SCOTCH instead.

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

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

I wish to thank all of the following people:

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

- Yves Secretan contributed to the MinGW32 port.

References


