Scotch and libScotch 3.4 User's Guide*

Research Report 1264-01

François Pellegrini
LaBRI, URM CNRS 5800
Université Bordeaux I
351 cours de la Libération, 33405 TALENCE, FRANCE
pelegrin@labri.u-bordeaux.fr

November 20, 2001

Abstract

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

*This work was supported by the GDR ARP of the French CNRS.

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 **6**
- 2.1 Description . . . 6
- 2.2 Availability . . . 6

3 Algorithms **6**
- 3.1 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 Execution scheme . . . 9
  - 3.1.6 Graph bipartitioning methods . . . 10
  - 3.1.7 Mapping onto variable-sized architectures . . . 12
- 3.2 Sparse matrix ordering by hybrid incomplete Nested Dissection . . . 13
  - 3.2.1 Minimum Degree . . . 13
  - 3.2.2 Nested Dissection . . . 13
  - 3.2.3 Hybridization . . . 13
  - 3.2.4 Performance criteria . . . 14
  - 3.2.5 Ordering methods . . . 14
  - 3.2.6 Graph separation methods . . . 15

4 Changes from version 3.3 **16**
- 4.1 Mapping onto variable-sized architectures . . . 16
- 4.2 Halo Approximate Minimum Fill algorithm . . . 16

5 Changes from version 3.2 **17**
- 5.1 File formats . . . 17
  - 5.1.1 Source graph files . . . 17
  - 5.1.2 Target architecture files . . . 17
- 5.2 Program interfaces . . . 17
  - 5.2.1 `map` . . . 17
  - 5.2.2 `ord` . . . 17

6 Files and data structures **17**
- 6.1 Source files . . . 18
  - 6.1.1 Old-style source files . . . 18
  - 6.1.2 New-style source files . . . 18
- 6.2 Geometry files . . . 19
- 6.3 Target files . . . 20
  - 6.3.1 Decomposition-defined architecture files . . . 20
  - 6.3.2 Algorithmically-coded architecture files . . . 21
  - 6.3.3 Variable-sized architecture files . . . 22
- 6.4 Mapping files . . . 23
- 6.5 Ordering files . . . 23
- 6.6 Vertex list files . . . 24

7 Programs **24**
7.1 Invocation . . . 24
7.2 Description . . . 26
7.2.1 `acpl` . . . 26
7.2.2 `amk_src` . . . 26
7.2.3 `amk_*` . . . 27
7.2.4 `atst` . . . 29
7.2.5 `map` . . . 29
7.2.6 `mtst` . . . 30
7.2.7 `ord` . . . 31
7.2.8 `out` . . . 33
7.2.9 `scv` . . . 36
7.2.10 `smk_*` . . . 37
7.2.11 `stst` . . . 37

8 Library **38**
8.1 Data formats . . . 38
8.1.1 Architecture format . . . 38
8.1.2 Graph format . . . 38
8.1.3 Block ordering format . . . 39
8.2 Strategy strings . . . 40
8.2.1 Mapping strategy strings . . . 40
8.2.2 Ordering strategy strings . . . 44
8.3 Target architecture handling routines . . . 49
8.3.1 `SCOTCH_archInit` . . . 49
8.3.2 `SCOTCH_archExit` . . . 49
8.3.3 `SCOTCH_archLoad` . . . 49
8.3.4 `SCOTCH_archSave` . . . 50
8.3.5 `SCOTCH_archBuild` . . . 50
8.3.6 `SCOTCH_archCmplt` . . . 51
8.3.7 `SCOTCH_archName` . . . 51
8.4 Graph handling routines . . . 52
8.4.1 `SCOTCH_graphInit` . . . 52
8.4.2 `SCOTCH_graphExit` . . . 52
8.4.3 `SCOTCH_graphLoad` . . . 52
8.4.4 `SCOTCH_graphSave` . . . 53
8.4.5 `SCOTCH_graphBuild` . . . 53
8.4.6 `SCOTCH_graphCheck` . . . 54
8.4.7 `SCOTCH_graphSize` . . . 54
8.4.8 `SCOTCH_graphData` . . . 55
8.5 High-level routines . . . 56
8.5.1 `SCOTCH_graphPart` . . . 56
8.5.2 `SCOTCH_graphMap` . . . 56
8.5.3 `SCOTCH_graphOrder` . . . 57
8.6 Strategy handling routines . . . 58
8.6.1 `SCOTCH_stratInit` . . . 58
8.6.2 `SCOTCH_stratExit` . . . 58
8.6.3 `SCOTCH_stratSave` . . . 58
8.6.4 `SCOTCH_stratBipart` . . . 59
8.6.5 `SCOTCH_stratMap` . . . 59
8.6.6 `SCOTCH_stratOrder` . . . 59

8.6.7 SCOTCH_graphSave 60
8.7 Mapping handling routines 60
8.7.1 SCOTCH_mapInit 60
8.7.2 SCOTCH_mapExit 61
8.7.3 SCOTCH_mapSave 61
8.7.4 SCOTCH_mapCompute 61
8.7.5 SCOTCH_mapView 62
8.8 Ordering handling routines 62
8.8.1 SCOTCH_orderInit 62
8.8.2 SCOTCH_orderExit 63
8.8.3 SCOTCH_orderSave 63
8.8.4 SCOTCH_orderCheck 64
8.9 Error handling routines 64
8.9.1 SCOTCH_errorPrint 64
8.9.2 SCOTCH_errorPrintW 64
8.9.3 SCOTCH_errorProg 65
8.10 Calling from Fortran 65
8.11 Compiling and linking 66

9 Installation **66**

10 Examples **66**

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 duration of all 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 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 and is never modified at run-time. 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, 18]. When the target machine is assumed to have a communication network in the shape of a complete graph, the static mapping problem turns into the *partitioning* problem, which has also been

intensely studied [4, 19, 27, 29, 44]. 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 [18, 51].

1.2 Sparse matrix ordering

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

An efficient way to compute fill reducing orderings of symmetric sparse matrices is to use recursive nested dissection [14]. It amounts to computing a vertex set S that separates the graph into two parts A and B, ordering S last, and proceeding recursively on parts A and B until their sizes become smaller than some threshold value. This ordering guarantees that no non-zero term can appear in the factorization process between unknowns of A and unknowns of B.

The main issue of the nested dissection ordering algorithm is thus to find small vertex separators that balance the remaining subgraphs as evenly as possible. Most often, vertex separators are computed by using direct heuristics [34, 25], or from edge separators [43, and included references] by minimum cover techniques [6, 26], but other techniques such as spectral vertex partitioning have also been used [44].

Provided that good vertex separators are found, the nested dissection algorithm produces orderings which, both in terms of fill-in and operation count, compare very favorably [17, 27, 42] to the ones obtained with the multiple minimum degree algorithm [37]. Moreover, the elimination trees induced by nested dissection are broader, shorter, and better balanced than the ones issued from multiple minimum degree [41], and therefore exhibit much more concurrency in the context of parallel Cholesky factorization [3, 11, 12, 16, 48, and included references].

1.3 Contents of this document

This document describes the capabilities and operations of Scotch, a software package devoted to static mapping, graph partitioning, and sparse matrix block ordering. Scotch allows the user to map efficiently any kind of weighted process graph onto any kind of weighted architecture graph, and provides high-quality block orderings of sparse matrices. The rest of this manual is organized as follows. Section 2 presents the goals of the Scotch project, and section 3 outlines the most important aspects of the mapping and ordering algorithms that we use. Section 4 summarizes the most important changes between version 3.4 and previous versions. Section 6 defines the formats of the files used in Scotch, section 7 describes the programs of the Scotch distribution, and section 8 defines the interface and operations of the libScotch library. Section 9 explains how to obtain and install the Scotch distribution. Finally, some practical examples are given in section 10.

2 The SCOTCH project

2.1 Description

SCOTCH is a project carried out at the *Laboratoire Bordelais de Recherche en Informatique* (LaBRI) of the Université Bordeaux I, by the ALiENor (ALgorithmics and ENvironments for parallel computing) team. 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 [38], all of which have been implemented in the SCOTCH software package [40]. More recently, 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 [41, 42].

2.2 Availability

The programs developed within the SCOTCH project, including the mapping, ordering, visualization, and graph building programs, are copyrighted by the LaBRI and their author. The SCOTCH distribution is publically available for research purposes, and may be licensed for commercial application. Section 9 describes how to obtain it via `WWW` or anonymous `ftp`.

3 Algorithms

3.1 Static mapping by Dual Recursive Bipartitioning

For a detailed description of the mapping algorithm and an extensive analysis of its performance, please refer to [38, 39]. In the next sections, we will only outline the most important aspects of the algorithm.

3.1.1 Static mapping

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

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

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

denotes the dilation of edge e_S, that is the number of edges of $E(T)$ used to route e_S.

3.1.2 Cost function and performance criteria

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

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

This function, which has already been considered by several authors for hypercube target topologies [8, 18, 22], 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 [18] on the CM-2, and by Hendrickson and Leland [23] 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, whose expressions are given below.

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

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

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

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

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

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

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

3.1.3 The Dual Recursive Bipartitioning algorithm

Our mapping algorithm uses a *divide and conquer* approach to recursively allocate subsets of processes to subsets of processors [38]. It starts by considering a set of processors, also called *domain*, containing all the processors of the target machine, and with which is associated the set of all the processes to map. At each step, the algorithm bipartitions a yet unprocessed domain into two disjoint subdomains, and calls a *graph bipartitioning algorithm* to split the subset of processes associated with the domain across the two subdomains, as sketched in the following.

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

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

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

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

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

- a *domain structure*, which represents a set of processors in the target architecture;
- a *domain bipartitioning function*, which, given a domain, bipartitions it in two disjoint subdomains;
- a *domain distance function*, which gives, in the target graph, a measure of the distance between two disjoint domains. Since domains may not be convex nor connected, this distance may be estimated. However, it must respect certain homogeneity properties, such as giving more accurate results as domain sizes decrease. The domain distance function is used by the graph bipartitioning algorithms to compute the communication function to minimize, since it allows the mapper to estimate the dilation of the edges that link vertices which belong to different domains. Using such a distance function amounts to considering that all routings will use shortest paths on the target architecture, which

is how most parallel machines actually do. We have thus chosen that our program would not provide routings for the communication channels, leaving their handling to the communication system of the target machine;

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

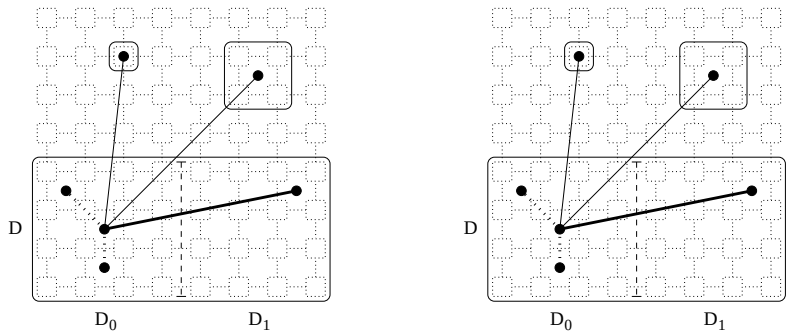
All these routines are seen as black-boxes by the mapping program, which can thus accept any kind of target architecture and process bipartitioning functions.

3.1.4 Partial cost function

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

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

which accounts for the dilation of edges internal to subgraph S' as well as for the one of edges which belong to the cocycle of S', as shown in figure 1. Taking into account the partial mapping results issued by previous bipartitionings makes it possible to avoid local choices that might prove globally bad, as explained below. This



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

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

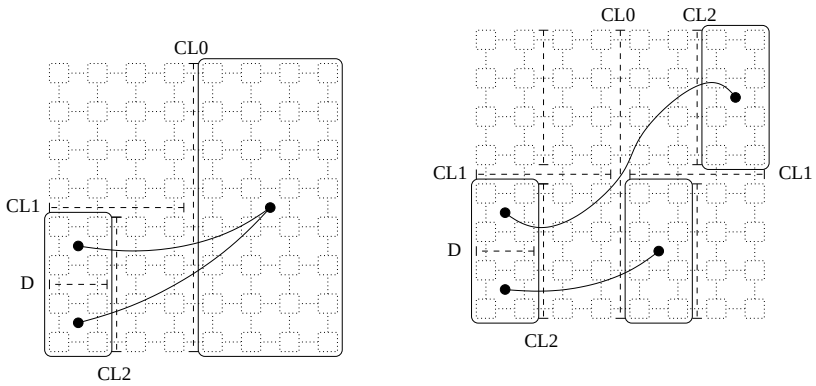
amounts to incorporate additional constraints to the standard graph bipartitioning problem, turning it into a more general optimization problem termed *skewed graph partitioning* by some authors [24].

3.1.5 Execution scheme

From an algorithmic point of view, our mapper behaves as a greedy algorithm, since the mapping of a process to a subdomain is never reconsidered, and at each step

of which iterative algorithms can be applied. The double recursive call performed at each step induces a recursion scheme in the shape of a binary tree, each vertex of which corresponds to a bipartitioning job, that is the bipartitioning of both a domain and its associated subgraph.

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



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

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

of depth-first and breadth-first sequencing schemes with respect to the structure of the source and target graphs is discussed in [39].

3.1.6 Graph bipartitioning methods

The core of our recursive mapping algorithm uses process graph bipartitioning methods as black boxes. It allows the mapper to run any type of graph bipartitioning method compatible with our criteria for quality. Bipartitioning jobs maintain an internal image of the current bipartition, indicating for every vertex of the job whether it is currently assigned to the first or 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. This permits us to define *mapping strategies*. The currently implemented graph bipartitioning methods are listed below.

Exactifier

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

Fiduccia-Mattheyses

The Fiduccia-Mattheyses heuristics [9] is an almost-linear improvement of the famous Kernighan-Lin algorithm [31]. It tries to improve the bipartition that is input to it by incrementally moving vertices between the subsets of the partition, as long as it can find sequences of moves that lower its communication cost. By considering sequences of moves instead of single swaps, the algorithm allows hill-climbing from local minima of the cost function. As an extension to the original Fiduccia-Mattheyses algorithm, we have developed new data structures, based on logarithmic indexings of arrays, that allow us to handle weighted graphs while preserving the almost-linearity in time of the algorithm [39].

As several authors quoted before [21, 28], the Fiduccia-Mattheyses algorithm gives better results when trying to optimize a good starting partition. Therefore, it should not be used on its own, but rather after greedy starting methods such as the Gibbs-Poole-Stockmeyer or the greedy-graph-growing methods.

Gibbs-Poole-Stockmeyer

This greedy bipartitioning method is inspired from an algorithm proposed by Gibbs, Poole, and Stockmeyer to minimize the dilation of graph orderings, that is the maximum absolute value of the difference between the numbers of neighbor vertices [15]. The graph is sliced by using a breadth-first spanning tree rooted at a randomly chosen vertex, and this process is iterated by selecting a new root vertex within the last layer as long as the number of layers increases. Then, starting from the current root vertex, vertices are assigned layer after layer to the first subdomain, until half of the total weight has been processed. Remaining vertices are then allocated to the second subdomain.

As for the original Gibbs, Poole, and Stockmeyer algorithm, it is assumed that the maximization of the number of layers results in the minimization of the sizes –and therefore of the cocycles– of the layers. This property has already been used by George and Liu to reorder sparse linear systems using the nested dissection method [14], and by Simon in [49].

Greedy-graph-growing

This greedy algorithm, which has been proposed by Karypis and Kumar [27], belongs to the GRASP ("*Greedy Randomized Adaptive Search Procedure*") class [32]. It consists in selecting an initial vertex at random, and repeatedly adding vertices to this growing subset, such that each added vertex results in the smallest increase in the communication cost function. This process, which stops when load balance is achieved, is repeated several times in order

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

Multi-level

This algorithm, which has been studied by several authors [4, 27, 20] and should be considered as a strategy rather than as a method since it uses other methods as parameters, repeatedly reduces the size of the graph to bipartition by finding matchings that collapse vertices and edges, computes a partition for the coarsest graph obtained, and projects the result back to the original graph, as shown in figure 3. The multi-level method, when used in conjunction

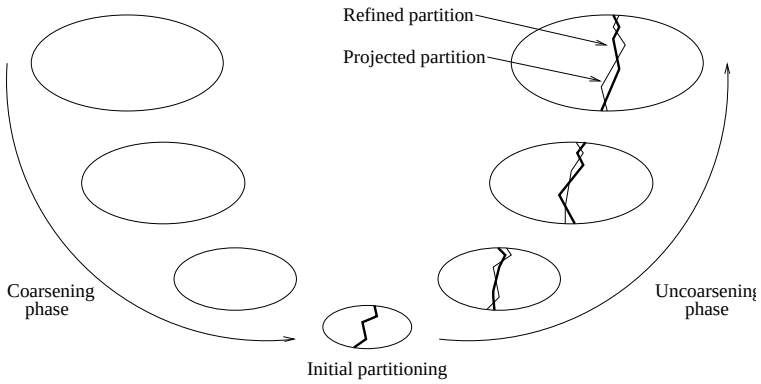


Figure 3: The multi-level partitioning process. In the uncoarsening phase, the light and bold lines represent for each level the projected partition obtained from the coarser graph, and the partition obtained after refinement, respectively.

with the Fiduccia-Mattheyses method to compute the initial partitions and refine the projected partitions at every level, usually leads to significant improvements in quality with respect to the plain Fiduccia-Mattheyses method. By coarsening the graph used by the Fiduccia-Mattheyses method to compute and project back the initial partition, the multi-level algorithm broadens the scope of the Fiduccia-Mattheyses algorithm, and makes possible for it to account for topological structures of the graph that would else be of a too high level for it to encompass in its local optimization process.

3.1.7 Mapping onto variable-sized architectures

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

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

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

3.2 Sparse matrix ordering by hybrid incomplete Nested Dissection

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

3.2.1 Minimum Degree

The Minimum Degree algorithm [50] is a local heuristic that performs its pivot selection by selecting from the graph a node of minimum degree.

The Minimum Degree algorithm is known to be a very fast and general purpose algorithm, and has received much attention over the last three decades (see for example [1, 13, 37]). However, the algorithm is intrinsically sequential, and very little can be theoretically proven about its efficiency.

3.2.2 Nested Dissection

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

Many theoretical results have been carried out on Nested Dissection ordering [5, 36], and its divide and conquer nature makes it easily parallelizable. The main issue of the nested dissection ordering algorithm is thus to find small vertex separators that balance the remaining subgraphs as evenly as possible. Most often, vertex separators are computed by using direct heuristics [34, 25], or from edge separators [43, and included references] by minimum cover techniques [6, 26], but other techniques such as spectral vertex partitioning have also been used [44].

In practice, Nested Dissection produces orderings which, both in terms of fill-in and operation count, compare well to the ones obtained with Minimum Degree [17, 27, 41]. Moreover, the elimination trees induced by nested dissection are broader, shorter, and better balanced, and therefore exhibit much more concurrency in the context of parallel Cholesky factorization [3, 11, 12, 17, 41, 48, and included references].

3.2.3 Hybridization

Due to their complementary nature, several schemes have been proposed to hybridize the two methods [25, 30, 41]. However, to our knowledge, only loose couplings have been achieved: incomplete Nested Dissection is performed on the graph to order, and the resulting subgraphs are passed to some Minimum Degree algorithm. This results in the fact that the Minimum Degree algorithm does not have exact degree values for all of the boundary vertices of the subgraphs, leading to a misbehavior of the vertex selection process.

Our ordering program implements a tight coupling of the Nested Dissection and Minimum Degree algorithms, that allows each of them to take advantage of the

information computed by the other. First, the Nested Dissection algorithm provides exact degree values for the boundary vertices of the subgraphs passed to the Minimum Degree algorithm (called *Halo* Minimum Degree since it has a partial visibility of the neighborhood of the subgraph). Second, the Minimum Degree algorithm returns the assembly tree that it computes for each subgraph, thus allowing for supervariable amalgamation, in order to obtain column-blocks of a size suitable for BLAS3 block computations.

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

The ordering program is completely parametrized by its ordering strategy. The nested dissection method allows the user to take advantage of all of the graph partitioning routines that have been developed in the earlier stages of the SCOTCH project. Internal ordering strategies for the separators are relevant in the case of sequential or parallel [16, 45, 46, 47] block solving, to select ordering algorithms that minimize the number of extra-diagonal blocks [5], thus allowing for efficient use of BLAS3 primitives, and to reduce inter-processor communication.

3.2.4 Performance criteria

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

3.2.5 Ordering methods

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

Multiple Minimum Degree
Multiple Minimum Degree method, as of [37].

Halo Approximate Minimum Degree
The Halo Approximate Minimum Degree method [42] is an improvement of the Aproximate Minimum Degree [1] algorithm, suited for use on subgraphs

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

produced by nested dissection methods. Its interest compared to classical minimum degree algorithms is that boundary vertices are processed using their real degree in the global graph rather than their (much smaller) degree in the subgraph, resulting in smaller fill-in and operation count. This method also implements amalgamation techniques that result in efficient block computations in the factoring and the solving processes.

Halo Approximate Minimum Fill
: The Halo Approximate Minimum Fill method is a variant of the Halo Approximate Minimum Degree algorithm, where the criterion to select the next vertex to permute is not based on its current estimated degree but on the minimization of the induced fill.

Graph Compression
: Graph compression method [2]. Compressing the graph amounts to merging cliques of vertices into single nodes, and has proven very useful to speed-up ordering as well as to improve the quality of separators, especially for stiffness matrices.

Degree reduction
: Degree reduction method. This method consists in selecting vertices whose degree is higher than a given multiple of the average degree and ordering them with the highest available indexes, the remaining subgraph being ordered by means of another strategy. The purpose of this algorithm is to remove from meshes vertices of very high degree, in order to speed-up separator building algorithms that make heavy use of adjacency lists.

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

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

Nested dissection
: Incomplete nested dissection method. Separators are computed recursively on subgraphs, and specific ordering methods are applied to the separators and to the resulting subgraphs (see section 3.2.3).

3.2.6 Graph separation methods

The core of our incomplete Nested Dissection algorithm uses graph separation methods as black boxes. It allows the orderer to run any type of graph separation method compatible with our criteria for quality, that is, reducing the size of the vertex separator while maintaining the loads of the separated parts within some user-specified tolerance. Separation jobs maintain an internal image of the current vertex separator, indicating for every vertex of the job whether it is currently assigned to one of the two parts, or to the separator. It is therefore possible to apply several different methods in sequence, each one starting from the result of the previous one, and to select the methods with respect to the job characteristics. This permits us to define *separation strategies*.

The currently implemented graph separation methods are listed below.

Fiduccia-Mattheyses
: This is a vertex-oriented version of the original, edge-oriented, Fiduccia-Mattheyses heuristics described in page 11.

Greedy-graph-growing
: This is a vertex-oriented version of the edge-oriented Greedy-graph-growing algorithm described in page 11.

Multi-level
: This is a vertex-oriented version of the edge-oriented multi-level algorithm described in page 12.

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

Vertex cover
: This algorithm computes a vertex separator by first computing an edge separator, that is, a bipartition of the graph, and then turning it into a vertex separator by using the method proposed by Pothen and Fang [43]. This method requires the computation of maximal matchings in the bipartite graphs associated with the edge cuts, which are built using Duff's variant [6] of the Hopcroft and Karp algorithm [26]. Edge separators are computed by using a bipartitioning strategy, which can use any of the graph bipartitioning methods described in section 3.1.6, page 10.

4 Changes from version 3.3

4.1 Mapping onto variable-sized architectures

SCOTCH now supports the mapping of graphs onto variable-sized architectures. This feature, which was originally only available through program `amk_src` for building decomposition-defined architectures from source graphs (see sections 7.2.2 and 6.3.1), is now available through the general mapping interface. SCOTCH can therefore be used as a general tool to provide high-quality, problem-dependant partitioning, whose characteristics are represented by the topology and the growth properties of the variable-sized architecture.

4.2 Halo Approximate Minimum Fill algorithm

The block ordering methods now include a Halo Approximate Minimum Fill algorithm, provided by Patrick Amestoy.

5 Changes from version 3.2

5.1 File formats

5.1.1 Source graph files

In order to speed-up I/O operations, as well as to provide a unified storage format across C and Fortran styles, a new source graph format has been designed. It is described in section 6.1.

At the time being, SCOTCH supports both the old-style and new-style file formats, but this may no longer be true in the future. Old-style source files should be converted into new-style files by means of the `scv` program, by specifying options `-Is` and `-Os`, such that an old-style source file is read as input and a new-style source file is produced on output (by default).

5.1.2 Target architecture files

The format of the compiled decomposition-defined ("`deco 1`") target architecture files has changed. All existing files of this type must be removed and regenerated from uncompiled files of type "`deco 0`" by using the new version of program `acpl`.

5.2 Program interfaces

5.2.1 `map`

The interface of program `map` has changed. The load imbalance ratio is no longer provided as a separate parameter by means of the `-b` option, but integrated within the mapping strategy as a parameter of the `f` (Fiduccia-Mattheyses) graph bipartitioning method. This has been done for the sake of coherence, since the load imbalance ratio was used only by this method, but also in order to enable the user to select different imbalance ratios in sub-expressions of the mapping strategy, if necessary.

5.2.2 `ord`

The interface of program `ord` has changed, too. Since the orderer is now able to order graphs using methods other than pure nested dissection, the two vertex separation (`-s`) and ordering (`-o`) strategies have been merged into a single ordering strategy, and the vertex separation strategy is now a parameter of the nested dissection ordering method.

As for the `map` program, the load imbalance ratio of the nested dissection method has been removed, and replaced by independent load imbalance ratios in the vertex-oriented (`f`) and edge-oriented (`e{strat=f}`) versions of the Fiduccia-Mattheyses algorithm.

6 Files and data structures

For the sake of portability, readability, and reduction of storage space, all the data files shared by the different programs of the SCOTCH project are coded in plain ASCII text exclusively. Although we may speak of "lines" when describing file formats, text-formatting characters such as newlines or tabulations are not mandatory, and are not taken into account when files are read. They are only used to provide

better readability and understanding. Whenever numbers are used to label objects, and unless explicitely stated, **numberings always start from zero**, not one.

6.1 Source files

Source files, which usually end in "`.src`", describe valuated graphs, and in particular the valuated process graphs to be mapped onto target architectures.

Source 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.

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

At the time being, both formats are supported (if the first data read is zero, it is assumed that the file is in the new style format, else the old style input routine is called). In the future, old-style compatibility will be removed.

The SCOTCH distribution comprises a file converter, `scv`, which automatically translates several graph formats, including the old-style source graph format, into the new-style source graph format. It is described in section 7.2.9.

6.1.1 Old-style source files

The first line of an old-style source file holds the number of vertices of the graph, followed by its number of arcs (which is twice the number of its edges). It is then followed by as many lines as there are vertices in the graph. Each of these lines begins with three numbers, precisely the label of the vertex, its load, and its degree (that is the number of arcs to neighboring vertices), followed by the description of these arcs. An arc is defined by two numbers, which represent the load of the edge, and the label of its other end vertex.

Vertices can be stored in any order in the file; similarly, the arcs of a given vertex can be in any order in its neighbor list. For example, figure 4 shows the contents of the old-style source file modeling a cube with unity vertex and edge weights. In this example, vertices are stored in order, but this is not mandatory.

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

Figure 4: Old-style source file representing a cube.

6.1.2 New-style source files

The first line of a new-style source file holds the source file version number, which is currently `0`. The second line holds the number of vertices of the graph, followed

by its number of arcs (that is, twice the number of its edges). The third line holds two figures: the graph base index value, 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 [21], 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 at the graph base index.

This header data is then followed by as many lines as there are vertices in the graph. 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 the label of its other end vertex.

If vertex labels are provided, vertices can be stored in any order in the file; similarly, the arcs of a given vertex can be in any order in its neighbor list. For example, figure 5 shows the contents of the new-style source file modeling a cube with unity vertex and edge weights.

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

Figure 5: New-style source file representing a cube.

6.2 Geometry files

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

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

Vertices can be stored in any order in the file. Moreover, a geometry file can have more coordinate data than there are vertices in the associated source file; only coordinates whose labels match labels of graph vertices will be taken into account. This

feature allows all subgraphs of a given source graph to share the same geometry file, provided that graph vertex labels remain unchanged. For example, figure 6 shows the contents of the 3D geometry file associated with the source graph of figure 5.

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

Figure 6: Geometry file associated with the source file of figure 5.

6.3 Target files

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

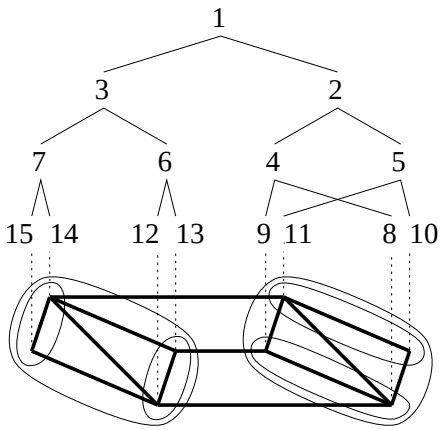
For space and time saving issues, some classical homogeneous architectures (2D and 3D meshes and tori, hypercubes, complete graphs, etc.) have been algorithmically coded within the mapper itself by the means of built-in functions. Instead of containing the whole graph decomposition data, their target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

6.3.1 Decomposition-defined architecture files

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

The "`deco 0`" header is followed by two integer numbers, which are the number of processors and the largest terminal number used in the decomposition, respectively. Two arrays follow. The first array has as many lines as there are processors. Each of these lines holds three numbers: the processor label, the processor weight (that is an estimation of its computational power), and its terminal number. The terminal number associated with every processor is obtained by giving the initial domain holding all the processors number 1, and by numbering the two subdomains of a given domain of number i with numbers $2i$ and $2i + 1$. The second array is a lower triangular diagonal-less matrix that gives the distance between all pairs of processors. This distance matrix, combined with the decomposition tree coded by terminal numbers, allows the evaluation by averaging of the distance between all pairs of domains. In order for the mapper to behave properly, distances between processors must be strictly positive numbers. Therefore, null distances are not accepted. For instance, figure 7 shows the contents of the architecture decomposition

file for UB(2, 3), the binary de Bruijn graph of dimension 3, as computed by the `amk_src` program.




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

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

6.3.2 Algorithmically-coded architecture files

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

`mesh2D` $dimX$ $dimY$
: Defines a bidimensional array of $dimX$ columns by $dimY$ rows. The vertex with coordinates $(posX, posY)$ has label $posY \times dimX + posX$.

`mesh3D` $dimX$ $dimY$ $dimZ$
: Defines a tridimensional array of $dimX$ columns by $dimY$ rows by $dimZ$ levels. The vertex with coordinates $(posX,posY,posZ)$ has label $(posZ \times dimY + posY) \times dimX + posX$.

`torus2D` $dimX$ $dimY$
: Defines a bidimensional array of $dimX$ columns by $dimY$ rows, with wraparound edges. The vertex with coordinates $(posX, posY)$ has label $posY \times dimX + posX$.

`torus3D` $dimX$ $dimY$ $dimZ$
: Defines a tridimensional array of $dimX$ columns by $dimY$ rows by $dimZ$ levels, with wraparound edges. The vertex with coordinates $(posX,posY,posZ)$ has label $(posZ \times dimY + posY) \times dimX + posX$.

`hcub` dim
: Defines a binary hypercube of dimension dim. The vertex labels are the decimal values of the binary representations of the vertex coordinates in the hypercube.

`cmplt` *size*

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

`leaf` *height cluster weight*

Defines a tree-leaf architecture with *height* levels and 2^{height} vertices. The tree-leaf graph models a machine whose topology is a complete binary tree, such that leaves are processors and all other nodes are communication routers, as shown in figure 8. Only the leaves are used to map processes, but distances between them are computed by considering the whole tree. This graph is used

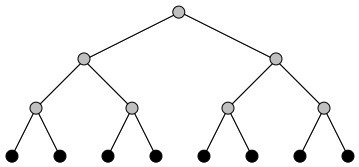


Figure 8: The "tree-leaf" graph of height 3. Processors are drawn in black and routers in grey.

to represent multi-stage machines with constant bandwidth, such as the CM-5 [33] for which experiments have shown that bandwidth is constant between every pair of processors and hardly depends on network congestion [35], or the SP-2 with power-of-two number of nodes.

The two additional parameters *cluster* and *weight* serve to model heterogeneous architectures for which multiprocessor nodes having several highly interconnected processors (typically by means of shared memory) are linked by means of networks of lower bandwidth. *cluster* represents the number of levels to traverse, starting from the root of the leaf, before reaching the multiprocessors, each multiprocessor having $2^{height-cluster}$ nodes. *weight* is the relative cost of extra-cluster links, that is, links in the upper levels of the tree-leaf graph. Links within clusters are assumed to have weight 1.

When there are no clusters at all, that is, in the case of purely homogeneous architectures, set *cluster* to be equal to *height*, and *weight* to 1.

6.3.3 Variable-sized architecture files

Variable-sized architectures are a class of algorithmically-coded architectures the size of which is not defined *a priori*. As for fixed-size algorithmically-coded architectures, they start with an abbreviation name of the architecture, followed by parameters specific to the architecture. The available built-in variable-sized architecture definitions are listed below.

`varcmplt`

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

`varhcub`

Defines a variable-sized hypercube. Domains are labeled such that the first domain is labeled 1, and the two subdomains of any domain i are labeled $2i$ and $2i + 1$. The distance between any two domains is the Hamming distance

between the common bits of the two domains, plus half of the absolute difference between the levels of the two domains, this latter term modeling the average distance on unknown bits. For instance, the distance between subdomain $9 = 1001_B$, of level 3 (since its leftmost 1 has been shifted left thrice), and subdomain $53 = 110101_B$, of level 5 (since its leftmost 1 has been shifted left five times), is 2: it is 1, which is the number of bits which differ between 1101_B (that is, $53 = 110101_B$ shifted rightwards twice) and 1001_B, plus 1, which is half of the absolute difference between 5 and 3.

6.4 Mapping files

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

Mapping files begin with the number of mapping lines which they contain, followed by that many mapping lines. Each mapping line holds a mapping pair, made of two integer numbers which are the label of a source graph vertex and the label of the target graph vertex onto which it is mapped. Mapping pairs can be stored in any order in the file; however, labels of source graph vertices must be all different. For example, figure 9 shows the result obtained when mapping the source graph of figure 5 onto the target architecture of figure 7. This one-to-one embedding of H(3) into UB(2, 3) has dilation 1, except for one hypercube edge which has dilation 3.

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

Figure 9: Mapping file obtained when mapping the hypercube source graph of figure 5 onto the binary de Bruijn architecture of figure 7.

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

6.5 Ordering files

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

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

Ordering files begin with the number of ordering lines which they contain, that is the number of vertices in the source graph, followed by that many ordering lines. Each ordering line holds an ordering pair, made of two integer numbers which are the label of a source graph vertex and its rank in the ordering. Ranks range from

zero to the number of vertices minus one. Ordering pairs can be stored in any order in the file; however, labels of source graph vertices must be all different.

For example, figure 10 shows the result obtained when reordering the source graph of figure 5.

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

Figure 10: Ordering file obtained when reordering the hypercube source graph of figure 5.

6.6 Vertex list files

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

Vertex lists are coded as lists of integer numbers. The first integer is the number of vertices in the list and the other integers are the labels of the selected vertices, given in any order. For example, figure 11 shows the list made from three vertices of labels 2, 45, and 7.

```
3   2   45   7
```

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

7 Programs

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

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

The general architecture of the SCOTCH project is given in figure 12.

7.1 Invocation

The programs comprising the SCOTCH project have been designed to run in command-line mode without any interactive prompting, so that they can be called easily from other programs by means of "`system ()`" or "`popen ()`" system calls,

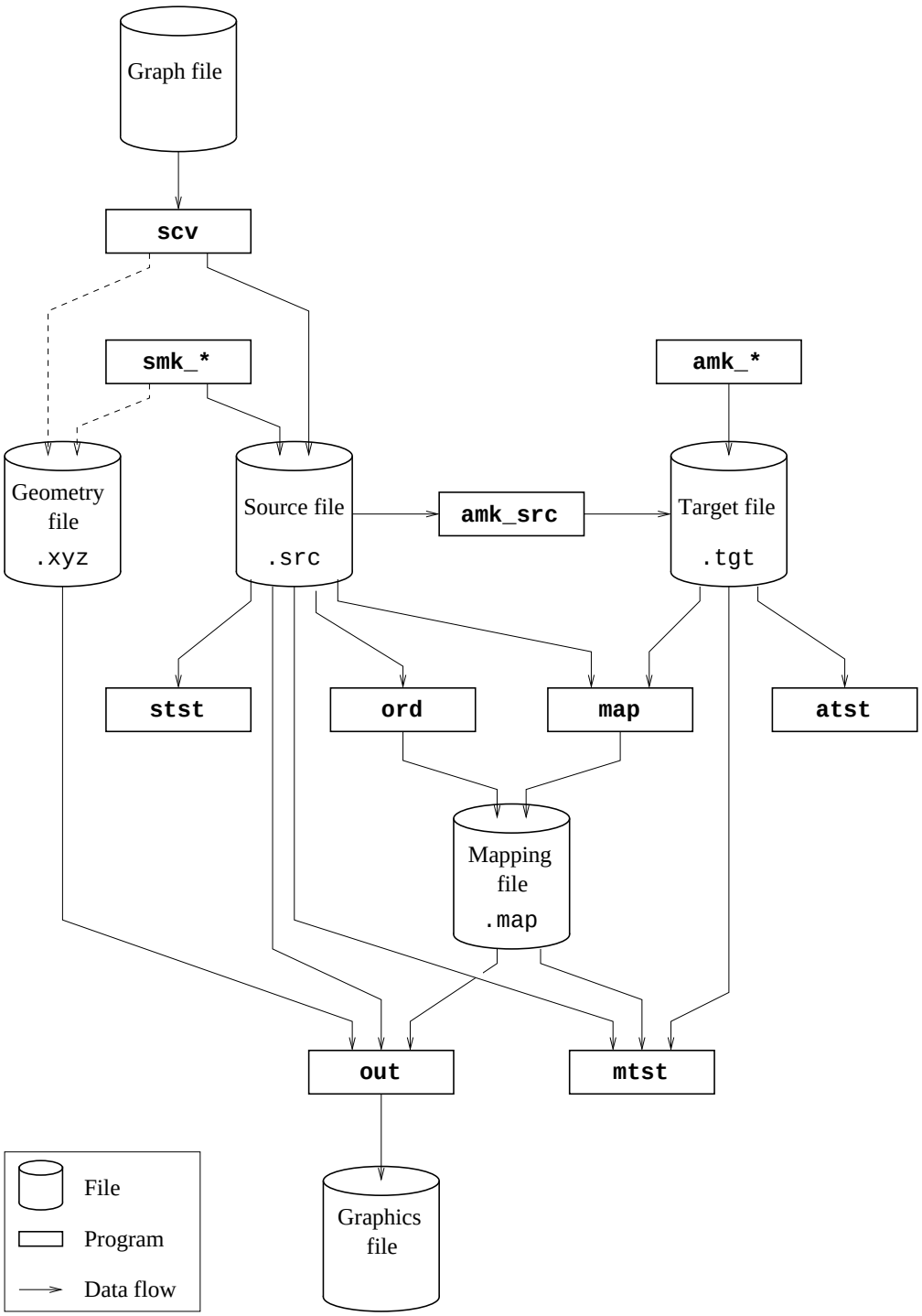


Figure 12: General architecture of the SCOTCH project.

or be piped together on a single shell command line. In order to facilitate this, whenever a stream name is asked for (either on input or output), the user may put a single "`-`" to indicate standard input or output. Moreover, programs read their input in the same order as stream names are given in the command line. It allows them to read all their data from a single stream (usually the standard input), provided that these data are ordered properly.

A brief on-line help is provided with all the programs. To get this help, use the "`-h`" option after the program name. The case of option letters is not significant, except when both the lower and upper cases of a letter have different meanings. When passing parameters to the programs, only the order of file names is significant; options can be put anywhere in the command line, in any order. Examples of use of the different programs of the SCOTCH project are provided in section 10.

Error messages are standardized, but may not be fully explanatory. However, most of the errors you may run into should be related to file formats, and located in "`...Load()`" routines. In this case, compare your data formats with the definitions given in section 6, and use the `stst` program to check the coherence of source graphs.

7.2 Description

7.2.1 `acpl`

Synopsis

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

Description

The program `acpl` is the decomposition-defined architecture file compiler. It processes architecture files of type "`deco 0`" built by hand or by the `amk_*` programs, to create a "`deco 1`" compiled architecture file of about four times the size of the original one; see section 6.3.1 for a detailed description of decomposition-defined target architecture file formats.
The mapper can read both original and compiled architecture file formats. However, compiled architecture files are read much more efficiently, as they are directly loaded into memory without further processing. Since the compilation time of a target architecture graph evolves as the square of its number of vertices, precompiling with `acpl` can save some time when many mappings are to be performed onto the same large target architecture.

Options

`-h` Display the program synopsis.

7.2.2 `amk_src`

Synopsis

`amk_src` [*input_source_file* [*output_target_file*]] *options*

Description

The program `amk_src` turns a source file into a decomposition-defined target file. It computes a recursive bipartitioning of the source graph, as well as the array of distances between all pairs of its vertices, both of which are combined

to give a decomposition-defined target architecture of same topology as the input source graph.

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

The `-b` option selects the recursive bipartitioning strategy used to build the decomposition of the source graph. For regular, unweighted, topologies, the `'-b(g|h)fx'` recursive bipartitioning strategy should work best. For irregular or weighted graphs, use the default strategy, which is more flexible. See also the manual page of function `SCOTCH_archBuild`, page 50, for further information.

Options

`-b`*strategy*
: Use recursive bipartitioning strategy *strategy* to build the decomposition of the architecture graph. The format of bipartitioning strategies is defined within section 8.2.1, at page 41.

`-h`
: Display the program synopsis.

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

7.2.3 `amk_*`

Synopsis

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

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

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

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

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

Description

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

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

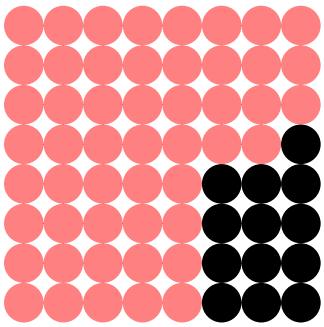
Program `amk_fft2` outputs the target architecture file of a binary Fast-Fourier-Transform graph of dimension *dim*. Vertex (l, m) of FFT(dim), with $0 \leq l \leq dim$ and $0 \leq m < 2^{dim}$, is linked to vertices $(l-1, m)$, $(l-1, m \bmod 2^{l-1})$, $(l+1, m)$, and $(l+1, m \oplus 2^l)$, if they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

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

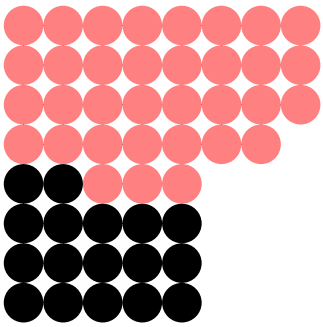
Program `amk_p2` outputs the target architecture file of a weighted path graph with two vertices, the weights of which are given as parameters.
This simple target topology is used to bipartition a source graph into two weighted parts with as few cut edges as possible. In particular, it is used to compute independent partitions of the processors of a multi-user parallel machine. As a matter of fact, if the yet unallocated part of the machine is represented by a source graph with n vertices, and n' processors are requested by a user in order to run a job (with $n' \leq n$), mapping the source graph onto the weighted path graph with two vertices of weights n' and $n - n'$ leads to a partition of the machine in which the allocated n' processors should be as densely connected as possible (see figure 13).

Options

`-h` Display the program synopsis.



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



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

Figure 13: Construction of partitions on a bidimensional 8×8 mesh architecture by weighted bipartitioning.

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

  `n` Nested dissection.

  `o` Dimension-per-dimension one-way dissection.

7.2.4 `atst`

Synopsis

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

Description

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

Options

`-h` Display the program synopsis.

7.2.5 `map`

Synopsis

`map` [*input_source_file* [*input_target_file* [*output_mapping_file* [*output_log_file*]]]] *options*

Description

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

The implemented mapping methods mainly derive from graph theory. In particular, graph geometry is never used, even if it is available; only topological

properties are taken into account. Mapping methods are used to define mapping strategies by means of selection, combination, grouping, and condition operators.
The only mapping method implemented in version 3.4 is the Dual Recursive Bipartitioning algorithm, which uses graph bipartitioning methods. Available bipartitioning methods include a multi-level algorithm that uses other bipartitioning methods to compute the initial and refined bipartitions, an improved implementation of the Fiduccia–Mattheyses heuristics designed to handle weighted graphs, a greedy method derived from the Gibbs, Poole, and Stockmeyer algorithm, the greedy-graph-growing heuristics, and a greedy "exactifying" refinement algorithm designed to balance vertex loads as much as possible; random and backtracking methods are also provided.
The `-m` option allows the user to define the mapping strategy.

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

Options

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

`-h` Display the program synopsis.

`-m`*strat*
: Apply mapping strategy *strat*. The format of mapping strategies is defined in section 8.2.1.

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

  `e` Keep edge weighs, if any.

  `v` Keep vertex weighs, if any.

`-V` Print the program version and copyright.

`-v`*verb*
: Set verbose mode to *verb*, which may contain several of the following switches. For a detailed description of the data displayed, please refer to the manual page of `mtst`.

  `m` Mapping information.

  `s` Strategy information. This parameter displays the default mapping strategy used by `map`.

  `t` Timing information.

7.2.6 `mtst`

Synopsis

`mtst` [*input_source_file* [*input_target_file* [*input_mapping_file* [*output_data_file*]]]] *options*

Description

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

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

Options

`-h` Display the program synopsis.

7.2.7 `ord`

Synopsis

`ord` [*input_source_file* [*output_ordering_file* [*output_log_file*]]] *options*

Description

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

Since its main purpose is to provide orderings that exhibit high concurrency for parallel block factorization, it comprises a nested dissection method [14], but classical [37] and state-of-the-art [1, 42] minimum degree algorithms are implemented as well. Ordering methods are used to define ordering strategies by means of selection, grouping, and condition operators.

For the nested dissection method, vertex separation methods comprise algorithms that directly compute vertex separators, as well as methods that build vertex separators from edge separators, *i.e.* graph bipartitions (all of the graph bipartitioning methods available in the static mapper `map` can be used in this latter case).
The `-o` option allows the user to define the ordering strategy.

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

Options

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

`-f`*output_source_file*
: Output the source graph resulting from the symbolic factorization of the reordered matrix to *output_source_file*. This source file may be processed by `out` (with option `-Om`) to show the pattern of the factored matrix.

`-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 leaves in the elimination tree. The structure of mapping files is given in section 6.4.

  When the geometry of the graph is available, this mapping file may be processed by `out` to display the vertex separators and supervariable amalgamations that have been computed.

`-o`*strat*
: Apply ordering strategy *strat*. The format of ordering strategies is defined in section 8.2.2.

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

`-v`*verb*
: Set verbose mode to *verb*, which may contain several of the following switches. For a detailed description of the data displayed, please refer to the manual page of `otst`.

  `o`
  : Ordering information, including fill-in and number of operations required to factor the system by Cholesky factorization (see section 3.2.4 for details on metrics).

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

  `t`
  : Timing information.

7.2.8 `out`

Synopsis

`out` [*input_source_file* [*input_geometry_file* [*input_mapping_file* [*output_visualization_file*]]]] *options*

Description

The `out` program is the graph, matrix, and mapping viewer program. It takes on input a source graph, its geometry file, and optionally a mapping result file, and produces a file suitable for display. At the time being, `out` can generate plain and encapsulated PostScript files for the display of adjacency matrix patterns and the visualization of planar graphs (although tridimensional objects can be displayed by means of isometric projection, the visualization of tridimensional mappings is not efficient), and OpenInventor files for the visualization of tridimensional graphs.
In the case of mapping visualization, the number of mapping pairs contained in the input mapping file may differ from the number of vertices of the input source graph; only mapping pairs the source labels of which match labels of source graph vertices will be taken into account for display. This feature allows the user to show the result of the mapping of a subgraph drawn on the whole graph, or else to outline the most important aspects of a mapping by restricting the display to a limited portion of the graph. For example, figure 14.b shows how the result of the mapping of a subgraph of the bidimensional mesh $M_2(4,4)$ onto the complete graph K(2) can be displayed on the whole $M_2(4,4)$ graph, and figure 14.c shows how the display of the same mapping can be restricted to a subgraph of the original graph.

Options

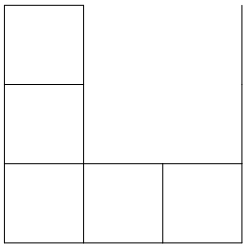
`-g`*parameters*
: Geometry parameters.

  `n` Do not read geometry data. This option can be used in conjunction with option `-om` to avoid reading the geometry file when displaying the pattern of the adjacency matrix associated with the source graph, since geometry data are not needed in this case. If this option is set, the geometry file is not read. However, if an *output_visualization_file* name is given in the command line, dummy *input_geometry_file* and *input_mapping_file* names must be specified so that the file argument count is correct. In this case, use the "`-`" parameter to take standard input as a dummy geometry input stream. In practice, the `-om` and `-gn` options always imply the `-mn` option.

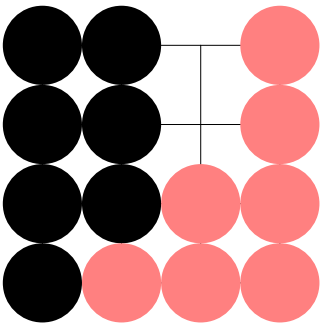
  `r` For bidimensional geometry only, rotate geometry data by 90 degrees, counter-clockwise.

`-h` Display the program synopsis.

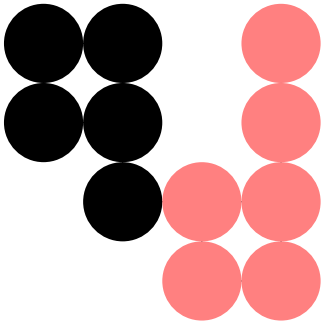
`-mn` Do not read mapping data, and display the graph without any mapping information. If this option is set, the mapping file is not read. However, if an *output_visualization_file* name is given in the command line, a dummy *input_mapping_file* name must be specified so that the file argument count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



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



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



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

Figure 14: PostScript visualization of a single mapping file with different subgraphs of the same source graph. Vertices covered with disks of the same color are mapped onto the same processor.

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

  `i` Output the graph in SGI's OpenInventor format, in ASCII mode, suitable for display by the `ivview` program of the IRIX distribution. The optional parameters are given below.

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

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

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

    `e` Encapsulated PostScript output, suitable for LaTeX use with `epsf`. Opposite of `f`.

    `f` Full-page PostScript output, suitable for direct printing. Opposite of `e`.

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

    `a` Avoid displaying the mapping disks. Opposite of `d`.

    `c` Color PostScript output. Opposite of `g`.

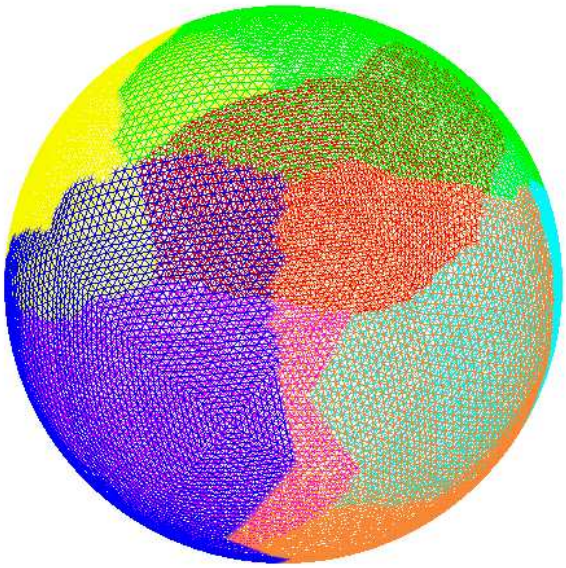


Figure 15: OpenInventor visualization of a sphere partitioned into 7 almost equal pieces by mapping onto the complete graph with 7 vertices. Vertices of same color are mapped onto the same processor.

`d` Display the mapping disks. Opposite of `a`.

`e` Encapsulated PostScript output, suitable for LaTeX use with `epsf`. Opposite of `f`.

`f` Full-page PostScript output, suitable for direct printing. Opposite of `e`.

`g` Grey-level PostScript output. Opposite of `c`.

`l` Large clipping. Mapping disks are included in the clipping area computation. Opposite of `s`.

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

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

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

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

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

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

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

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

7.2.9 scv

Synopsis

scv [*input_graph_file* [*output_source_file* [*output_geometry_file*]]] *options*

Description

The program scv is the source graph converter. It takes on input a graph file of the format specified with the -i option, and outputs its equivalent in the format specified with the -o option, along with its associated geometry file whenever geometrical data is available. At the time being, it only accepts five external input formats: the CHACO graph format [21], S. W. Hammond's format, H. D. Simon's simple format, the Harwell-Boeing collection format [7], and Structural Dynamics Research's Universal Data Set 2412 format. Two output format are provided: our SCOTCH source graph and geometry data format, and the CHACO graph format.

Options

-h Display the program synopsis.

-i*format*
: Specify the type of input graph to be converted into our source graph format. The available input formats are listed below.

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

  c CHACO v1.0 format.

  h Steve Hammond's graph format.

  s SCOTCH source graph format.

  t Horst Simon's simple graph format.

  u Universal Data Set 2412 format. On output, this node/element structure is turned into a communication graph such that vertices represent elements and there exists an edge between two vertices if the two end elements share a node in the original UDS graph. Since UDS format files are tag files, they do not have a well defined end. Therefore, one cannot append data of different nature to the input stream used to read this graph, since it will make the graph loading routine to fail.

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

  c CHACO v1.0 format.

  s SCOTCH source graph format.

Default option set is "-Ib0 -Os".

7.2.10 smk_*

Synopsis

smk_hy *dim* [*output_source_file*] *options*

smk_m2 *dimX* [*dimY* [*output_source_file*]] *options*

smk_ub2 *dim* [*output_source_file*] *options*

Description

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

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

Program smk_m2 outputs the source file of a bidimensional mesh with *dimX* columns and *dimY* rows. If the -t option is set, tori are built instead of meshes. The vertex of coordinates $(posX, posY)$ is labeled $posY \times dimX + posX$.

Program smk_ub2 outputs the source file of a binary unoriented de Bruijn graph of dimension *dim*. Vertices are labeled according to the decimal value of their binary representation.

Options

-g*output_geometry_file*
: Output graph geometry to file *output_geometry_file* (for smk_m2 only). As for all other file names, "-" may be used to indicate standard input.

-h
: Display the program synopsis.

-t
: Build a torus rather than a mesh (for smk_m2 only).

7.2.11 stst

Synopsis

stst [*input_source_file* [*output_data_file*]] *options*

Description

The program stst is the source graph tester. It checks the coherence 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.

Options

`-h` Display the program synopsis.

8 Library

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

- Source graph handling routines, that serve to declare, build, load, save, and check the consistency of source graphs.
- Target architecture handling routines, that allow the user to declare, build, load, and save target architectures.
- Strategy handling routines, that allow the user to declare and build mapping and ordering strategies.
- Mapping routines, that serve to declare, compute, and save mappings of source graphs to target architectures by means of mapping strategies.
- Ordering routines, that allow the user to declare, compute, and save orderings of source graphs by means of ordering strategies.
- Error handling routines, that allow the user either to provide his own error servicing routines, or to use the default routines provided in the LIBSCOTCH distribution.

All of the routines of the LIBSCOTCH library are prefixed with "`SCOTCH_`". The remainder of the function name is made of the name of the type of object to which the functions apply (e.g. "`graph`", "`target`", "`map`", etc.), followed by the type of action performed on this object: "`Init`" for the initialization of the object, "`Exit`" for the freeing of its internal structures, "`Load`" for loading the object from a stream, etc.

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

8.1 Data formats

All of the data used in the LIBSCOTCH interface are of integer type `SCOTCH_Num`.

8.1.1 Architecture format

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

8.1.2 Graph format

Source graphs are described by means of adjacency lists. The description of a graph requires several `SCOTCH_Num` scalars and arrays, as shown in figure 16. They have the following meaning:

`baseval`
: Base value for all array indexings.

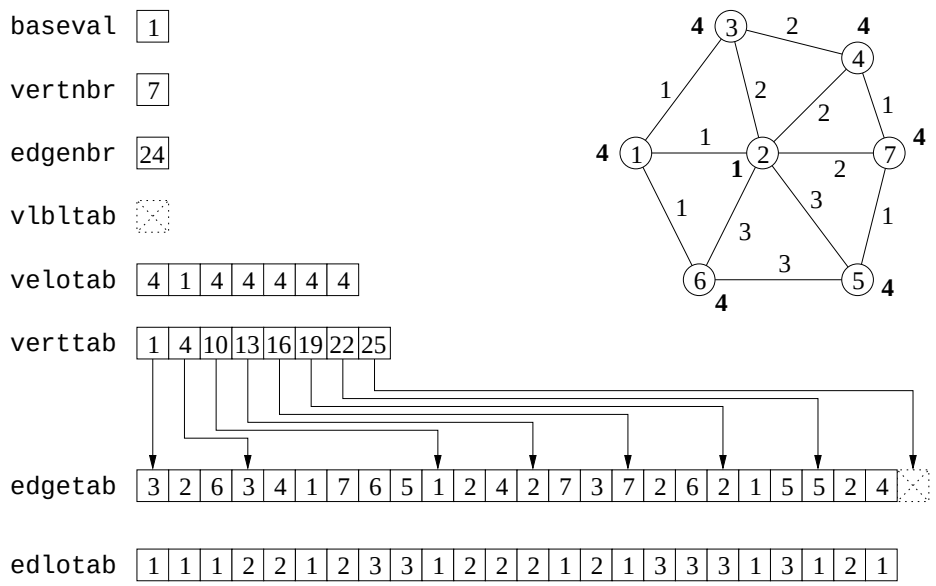


Figure 16: Sample graph and its description by LIBSCOTCH arrays. Numbers in vertices are vertex indices, bold numbers close to vertices are vertex loads, and numbers close to edges are edge loads.

`vertnbr`
: Number of vertices in graph.

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

`verttab`
: Vertex degree array and index into `edgetab`. `verttab[0]` is always equal to `baseval`, and `verttab[vertnbr]` is always equal to `edgenbr` + `baseval`.

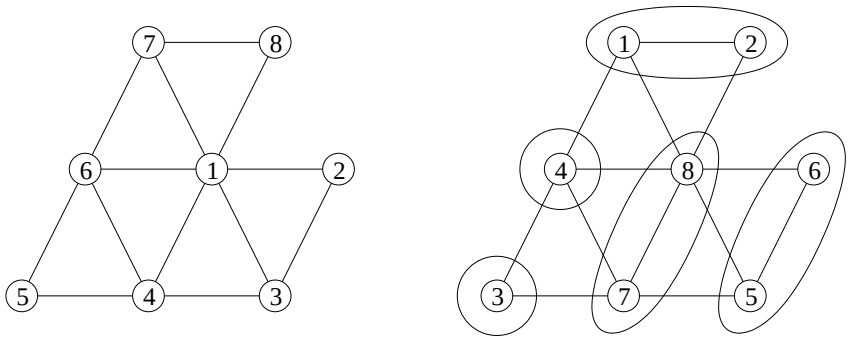
`velotab`
: Array, of size `vertnbr`, holding the integer load associated with each vertex.

8.1.3 Block ordering format

Block orderings associated with graphs are described by means of blocking and permutation arrays, made of `SCOTCH_Nums`, as shown in figure 17. They have the following meaning:

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

`rangtab`
: Array of ranges for the column blocks. Column block c, with $0 \leq c <$ `cblknbr`, contains columns with indices ranging from `rangtab[`i`]` to `rangtab[`$i+1$`]`, exclusive, in the reordered matrix. Therefore, `rangtab[0]` is always equal to `baseval`, and `rangtab[cblknbr]` is always equal to `vertnbr` + `baseval`, where `vertnbr` is the number of columns, that is, the number of vertices in the associated graph. In order to avoid memory errors when column blocks



`cblknbr`	5
`rangtab`	1 3 4 5 7 9
`permtab`	8 6 5 7 3 4 1 2
`peritab`	7 8 5 6 3 2 4 1

Figure 17: Sample block ordering and its description by LIBSCOTCH arrays.

are all single columns, the size of `rangtab` must always be one more than the number of columns, that is, `vertnbr` + 1.

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

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

All of the indices used in arrays are based according to the base value of the graph to which the ordering refers.

8.2 Strategy strings

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

8.2.1 Mapping strategy strings

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

`b`
: Dual Recursive Bipartitioning mapping algorithm, as defined in section 3.1.3. The parameters of the DRB mapping method are listed below.

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

`t` Tie job pools together. Subjobs are stored in same pool as their parent job.

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

`map=`*tie*

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

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

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

`poli=`*policy*

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

`L` Most neighbors of higher level.

`l` Highest level.

`r` Random.

`S` Most neighbors of smaller size.

`s` Biggest size.

`strat=`*strat*

Apply bipartitioning strategy *strat* to each bipartitioning job. A bipartitioning strategy is made of one or several bipartitioning methods, which can be combined by means of strategy operators. Strategy operators are listed below, by increasing precedence.

strat1 `|` *strat2*

Selection operator. The result of the selection is the best bipartition of the two that are obtained by the separate application of *strat1* and *strat2* to the current bipartition.

strat1 strat2

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

`(`*strat*`)`

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

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

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

cond1 `|` *cond2*

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

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

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

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

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

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

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

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

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

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

The currently available graph bipartitioning methods are the following.

`b`
: Backtracking method.

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

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

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

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

`type=`*type*
: Set the type of vertices that are considered for moving between partitions. *type* is `a` when all vertices are considered, and `b` when only vertices belonging to the current boundary may be selected. Selecting boundary vertices only is useful for weighted graphs when the obtainment of connected subgraphs is essential, as it is the case for instance when using the mapper as a domain decomposer for parallel Conjugate Gradient methods.

`g`
: Gibbs-Poole-Stockmeyer method.

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

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

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

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

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

`rat=`*rat*
: Set the threshold maximum coarsening ratio over which graphs are no longer coarsened. The ratio of any given coarsening cannot be less that 0.5 (case of a perfect matching), and cannot be greater than 1.00. Coarsening stops when either the coarsening ratio is above the maximum coarsening ratio, or the graph has fewer vertices than the minimum number of vertices allowed.

`type=`*type*
: Set the type of matching that is used to coarsen the graphs. *type* is `h` for heavy-edge matching, `l` for light-edge matching, or `r` for random matching.

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

`x`
: Exactifying method.

`z` Zero method. This method moves all of the vertices to the first part. Its main use is to stop the bipartitioning process, if some condition is true, when mapping onto variable-sized architectures (see section 3.1.7).

8.2.2 Ordering strategy strings

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

(*strat*)
: Grouping operator. The strategy enclosed within the parentheses is treated as a single ordering method.

/*cond*?*strat1*[:*strat2*];
: Condition operator. According to the result of the evaluation of condition *cond*, either *strat1* or *strat2* (if it is present) is applied. The condition applies on the characteristics of the current node of the separation 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.

    `desc`
    : The number of descendents of the current node. If this variable is zero, then the node is a leaf, else it is a separator. Integer.

    `level`
    : The level of the node in the separation tree, starting from zero at the root of the tree (which is either the first separator or the entire graph if no separation occured). Integer.

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

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

The currently available graph ordering methods are the following.

`a` Approximate Minimum Degree method of [1], provided by its authors.

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

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

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

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

`d`
: Degree reduction method. The parameters of the degree reduction method are listed below.

`rat=`*rat*
: Multiple of the average degree over which vertices are removed from the graph and ordered separately with the highest indexes available.

`strat=`*strat*
: Ordering strategy to apply to the remaining subgraph.

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

`halo=`[`y`—`n`]
: Indicate if subgraph halo information should be used or not. Using the halo leads to better orderings, at the expense of higher computation time.

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

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

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

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

`h`
: Block Halo Approximate Minimum Degree method [42]. The parameters of the Halo Approximate Minimum Degree method are listed below.

`halo=`[`y`—`n`]
: Indicate if subgraph halo information should be used or not. Using the halo leads to better orderings, at the expense of higher computation time.

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

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

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

`m`
: Multiple Minimum Degree method, as of [37], provided by Alex Pothen. The Block Halo Approximate Minimum Degree method should be preferred.

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

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

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

  `sep=`*strat*
  : Set the node separation strategy that is used on every leaf of the separation tree to make it grow. A node separation strategy is made of one or several node separation methods, which can be combined by means of strategy operators. Strategy operators are listed below, by increasing precedence.

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

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

    `(`*strat*`)`
    : Grouping operator. The strategy enclosed within the parentheses is treated as a single separation method.

    `/`*cond*`?`*strat1*`[:`*strat2*`];`
    : Condition operator. According to the result of the evaluation of condition *cond*, either *strat1* or *strat2* (if it is present) is applied. The condition applies on the characteristics of the current subgraph,

and can be built from logical and relational operators. Conditional operators are listed below, by increasing precedence.

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

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

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

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

  `level`
  : The level of the node in the separation tree, starting from zero at the root of the tree. Integer.

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

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

The currently available vertex separation methods are the following.

`e` Edge-separation method. This method builds vertex separators from edge separators, by the method proposed by Pothen and Fang [43], which uses a variant of the Hopcroft and Karp algorithm due to Duff [6]. This method is expensive and most often yields poorer results than direct vertex-oriented methods such as the vertex vertex Greedy-graph-growing and the vertex Fiduccia-Mattheyses algorithms. The parameters of the edge-separation method are listed below.

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

`strat`=*strat*
: Set the graph bipartitioning strategy that is used to compute the edge bipartition. The syntax of bipartitioning strategy strings is defined within section 8.2.1, at page 41.

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

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

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

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

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

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

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

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

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

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

`rat=`*rat*
: Set the threshold maximum coarsening ratio over which graphs are no longer coarsened. The ratio of any given coarsening cannot be less that 0.5 (case of a perfect matching), and cannot be greater than 1.00. Coarsening stops when either the coarsening ratio is above the maximum coarsening ratio, or the graph has fewer vertices than the minimum number of vertices allowed.

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

`t` Thinner method. This method quickly eliminates all useless vertices of the current separator. It searches the separator for vertices that have no neighbors

in one of the two parts, and moves these vertices to the part they are connected to. This method may be used to refine separators during the uncoarsening phase of the multi-level method, and is quicker than a vertex Fiduccia-Mattheyses algorithm with {`move=0`}.

`z` Zero method. This method moves all of the vertices to the first part, with an empty separator. Its main use is to stop the vertex bipartitioning process if some condition is true.

8.3 Target architecture handling routines

8.3.1 `SCOTCH_archInit`

Synopsis

```
int SCOTCH_archInit (SCOTCH_Arch *  archptr);
```

Description

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

Return values

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

8.3.2 `SCOTCH_archExit`

Synopsis

```
void SCOTCH_archExit (SCOTCH_Arch *  archptr);
```

Description

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

8.3.3 `SCOTCH_archLoad`

Synopsis

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

Description

The SCOTCH_archLoad routine loads the contents of the SCOTCH_Arch structure pointed to by archptr with the source graph description available from stream stream in the Scotch target architecture format (see section 6.3).

Return values

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

8.3.4 SCOTCH_archSave

Synopsis

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

Description

The SCOTCH_archSave routine saves the contents of the SCOTCH_Arch structure pointed to by archptr to stream stream, in the Scotch target architecture format (see section 6.3).

Return values

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

8.3.5 SCOTCH_archBuild

Synopsis

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

Description

The SCOTCH_archBuild routine fills the architecture structure pointed to by archptr with the decomposition-defined target architecture computed by applying the graph bipartitioning strategy pointed to by straptr to the architecture graph pointed to by grafptr.

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

to (vertnbr − 1) if the graph base is 0, and from 1 to vertnbr if the graph base is 1.

Bipartitioning strategies are declared by means of the SCOTCH_stratBipart function, described in page 59. The syntax of bipartitioning strategy strings is defined within section 8.2.1, at page 41. Additional information may be obtained from the manual page of amk_src, the stand-alone executable that uses function SCOTCH_archBuild to build decomposition-defined target architecture from source graphs, at page 26.

Return values

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

8.3.6 SCOTCH_archCmplt

Synopsis

```
int SCOTCH_archCmplt (SCOTCH_Arch *      archptr,
                      const SCOTCH_Num   procnbr);
```

Description

The SCOTCH_archCmplt routine fills the SCOTCH_Arch structure pointed to by archptr with the description of a complete graph architecture with procnbr processors.

Return values

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

8.3.7 SCOTCH_archName

Synopsis

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

Description

The SCOTCH_archName function returns a string containing the name of the architecture pointed to by archptr. Since Fortran routines cannot return string pointers, the SCOTCHFARCHNAME routine takes as second and third parameters a CHARACTER() array to be filled with the name of the architecture, and the INTEGER size of the array, respectively, the fourth parameter being the return value.

Return values

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

8.4 Graph handling routines

8.4.1 SCOTCH_graphInit

Synopsis

```
int SCOTCH_graphInit (SCOTCH_Graph *  grafptr);
```

Description

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

Return values

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

8.4.2 SCOTCH_graphExit

Synopsis

```
void SCOTCH_graphExit (SCOTCH_Graph *  grafptr);
```

Description

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

8.4.3 SCOTCH_graphLoad

Synopsis

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

Description

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

To ease the handling of source graph files by programs written in C as well as in Fortran, The base value of the graph 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 by `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 the is specified in the `stream` data.

Return values

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

8.4.4 SCOTCH_graphSave

Synopsis

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

Description

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

Return values

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

8.4.5 SCOTCH_graphBuild

Synopsis

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

Description

The SCOTCH_graphBuild routine fills the source graph structure pointed to by grafptr with all of the data that is passed to it.

vertnbr is the number of vertices, verttab is the adjacency index array, of size (vertnbr+1), velotab is the vertex load array, of size vertnbr, vlbltab is the vertex label array, of size vertnbr, edgenbr is the number of arcs (that is, twice the number of edges), edgetab is the adjacency array, of size edgenbr, edlotab is the arc load array, of size edgenbr, baseval is the graph base value for index arrays (typically 0 for C-based structures and 1 for Fortran-based structures), flagval is a combination of the following integer values, that may be added or bitwise-ored:

0 Accont for vertex and edge weights in all of the algorithms that will involve the source graph.

1 Do not account for vertex weights.

2 Do not account for edge weights.

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

Return values

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

8.4.6 SCOTCH_graphCheck

Synopsis

```
int SCOTCH_graphCheck (SCOTCH_Graph *  grafptr);
```

Description

The SCOTCH_graphCheck routine checks the consistency of the given SCOTCH_Graph structure pointed to by grafptr.

Return values

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

8.4.7 SCOTCH_graphSize

Synopsis

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

Description

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

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

8.4.8 SCOTCH_graphData

Synopsis

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

Description

The SCOTCH_graphData routine is the opposite of the SCOTCH_graphBuild routine. It is a multiple accessor that returns scalar values and array references.

vertptr is the pointer to a location that will hold the number of vertices, verttab is the pointer to a location that will hold the reference to the adjacency index array, of size (*vertptr + 1), velotab is the pointer to a location that will hold the reference to the vertex load array, of size *vertptr, vlbltab is the pointer to a location that will hold the reference to the vertex label array, of size vertnbr, edgeptr is the pointer to a location that will hold the number of arcs (that is, twice the number of edges), edgetab is the pointer to a location that will hold the reference to the adjacency array, of size edgenbr, edlotab is the pointer to a location that will hold the reference to the arc load array, of size edgenbr, baseptr is the pointer to a location that will hold the graph base value for index arrays (typically 0 for C-based structures and 1 for Fortran-based structures), flagptr is the pointer to a location that will hold the a combination of flag integer values, that may be added or bitwise-ored, and are defined in the manual pages of function SCOTCH_graphBuild, page 54.

8.5 High-level routines

These routines provide high-level functionalities and free the user from the burden of calling in sequence several of the low-level SCOTCH_map and SCOTCH_order routines described in the following sections.

8.5.1 SCOTCH_graphPart

Synopsis

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

Description

The SCOTCH_graphPart routine computes a partition of the source graph structure pointed to by grafptr in partnbr parts, using the graph partitioning strategy pointed to by stratptr, and returns the partition data in the array pointed to by parttab.

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

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

Return values

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

8.5.2 SCOTCH_graphMap

Synopsis

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

Description

The SCOTCH_graphMap routine computes a mapping of the source graph structure pointed to by grafptr onto the target architecture pointed to by archptr, using the mapping strategy pointed to by stratptr, and returns the mapping data in the array pointed to by parttab.

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

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

Return values

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

8.5.3 `SCOTCH_graphOrder`

Synopsis

```
int SCOTCH_graphOrder (const SCOTCH_Graph *  grafptr,
                       const SCOTCH_Strat *  straptr,
                       SCOTCH_Num *          cblkptr,
                       SCOTCH_Num *          rangtab,
                       SCOTCH_Num *          permtab,
                       SCOTCH_Num *          peritab);
```

Description

The `SCOTCH_graphOrder` routine computes a block ordering of the unknowns of the symmetric sparse matrix whose adjacency structure is represented by the source graph structure pointed to by `grafptr`, using the ordering strategy pointed to by `stratptr`, and returns ordering data in the scalar pointed to by `cblkptr` and the three arrays `rangtab`, `permtab`, and `peritab`.

The `rangtab`, `permtab`, and `peritab` arrays should have been previously allocated, of a size sufficient to hold as many `SCOTCH_Num` integers as there are vertices in the source graph, plus one in the case of `rangtab`.

On return, `permtab` holds the direct permutation of the unknowns, that is, vertex i of the original graph has index `permtab[i]` in the reordered graph, while `peritab` holds the inverse permutation: vertex i in the reordered graph had index `peritab[i]` in the original graph. All of these indices are numbered according to the base value of the source graph: permutation indices are numbered from `baseval` to `vertnbr` + `baseval` − 1, that is, from 0 to (`vertnbr` − 1) if the graph base is 0, and from 1 to `vertnbr` if the graph base is 1.

The two other result fields, `*cblkptr` and `rangtab`, contain blocking data: `*cblkptr` holds the number of column blocks of the produced ordering, and `rangtab` holds the starting indices of each of the column blocks, in increasing order, so that column block i starts at index `rangtab[i]` and ends at index `(rangtab[i + 1] - 1)`, inclusive, in the new ordering. Indices in `rangtab` are based, in the same way as for `permtab` and `peritab`. Therefore, `rangtab[0]` = `baseval`, and `rangtab[*cblkptr]` = `vertnbr` + `baseval`.

Return values

`SCOTCH_graphOrder` returns 0 if the partition of the graph has been successfully computed, and 1 else. In this last case, the `rangtab`, `permtab`, and

peritab arrays may however have been partially or completely filled, but their contents are not significant.

8.6 Strategy handling routines

8.6.1 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 graph data is no longer of use, call function SCOTCH_stratExit to free its internal structures.

Return values

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

8.6.2 SCOTCH_stratExit

Synopsis

```
void SCOTCH_stratExit (SCOTCH_Strat *  archptr);
```

Description

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

8.6.3 SCOTCH_stratSave

Synopsis

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

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.

Return values

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

8.6.4 SCOTCH_stratBipart

Synopsis

```
int SCOTCH_stratBipart (SCOTCH_Strat *  straptr,
                        const char *    string);
```

Description

The SCOTCH_stratBipart routine fills the strategy structure pointed to by straptr with the bipartitioning strategy string pointed to by string. From this point, strategy strat can only be used as a bipartitioning strategy, to be used by function SCOTCH_graphBipart, for instance.

The array of characters pointed to by string must be null-terminated.

Return values

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

8.6.5 SCOTCH_stratMap

Synopsis

```
int SCOTCH_stratMap (SCOTCH_Strat *  straptr,
                     const char *    string);
```

Description

The SCOTCH_stratMap routine fills the strategy structure pointed to by straptr with the mapping strategy string pointed to by string. From this point, strategy strat can only be used as a mapping strategy, to be used by function SCOTCH_graphMap, for instance.

The array of characters pointed to by string must be null-terminated.

Return values

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

8.6.6 SCOTCH_stratOrder

Synopsis

```
int SCOTCH_stratOrder (SCOTCH_Strat *  straptr,
                       const char *    string);
```

Description

The SCOTCH_stratOrder routine fills the strategy structure pointed to by straptr with the ordering strategy string pointed to by string. From this point, strategy strat can only be used as an ordering strategy, to be used by function SCOTCH_graphOrder, for instance.

The array of characters pointed to by string must be null-terminated.

Return values

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

8.6.7 SCOTCH_graphSave

Synopsis

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

Description

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

Return values

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

8.7 Mapping handling routines

8.7.1 SCOTCH_mapInit

Synopsis

```
int SCOTCH_mapInit (SCOTCH_Mapping *       mappptr,
                    const SCOTCH_Graph *   grafptr,
                    const SCOTCH_Arch *    archptr,
                    SCOTCH_Num *           mapptab);
```

Description

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

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

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

Return values

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

8.7.2 SCOTCH_mapExit

Synopsis

```
void SCOTCH_mapExit (SCOTCH_Mapping *  mappptr);
```

Description

The SCOTCH_mapExit function frees the contents of a SCOTCH_Mapping structure previously initialized by SCOTCH_mapInit. All following calls to SCOTCH_map routines other than SCOTCH_mapInit, using this structure as parameter, may yield unpredictable results.

8.7.3 SCOTCH_mapSave

Synopsis

```
int SCOTCH_mapSave (const SCOTCH_Mapping *  mappptr,
                    FILE *                  stream);
```

Description

The SCOTCH_mapSave routine saves the contents of the SCOTCH_Mapping structure pointed to by mappptr to stream stream, in the Scotch mapping format (see section 6.4).

Return values

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

8.7.4 SCOTCH_mapCompute

Synopsis

```
int SCOTCH_mapCompute (SCOTCH_Mapping *       mappptr,
                       const SCOTCH_Strat *   straptr);
```

Description

The SCOTCH_mapCompute routine computes a mapping on the given SCOTCH_Mapping structure pointed to by mappptr using the mapping strategy pointed to by stratptr.

On return, every cell of the mapping array (see section 8.7.1) holds the number of the target vertex to which the corresponding source vertex is mapped. The numbering of target values is *not* based: target vertices are numbered from 0 to the number of target vertices, minus 1.

Return values

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

8.7.5 SCOTCH_mapView

Synopsis

```
int SCOTCH_mapView (const SCOTCH_Mapping *  mappptr,
                    FILE *                  stream);
```

Description

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

Return values

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

8.8 Ordering handling routines

8.8.1 SCOTCH_orderInit

Synopsis

```
int SCOTCH_orderInit (SCOTCH_Ordering *      ordeptr,
                      const SCOTCH_Graph *   grafptr,
                      SCOTCH_Num *           cblkptr,
                      SCOTCH_Num *           rangtab,
                      SCOTCH_Num *           permtab,
                      SCOTCH_Num *           peritab);
```

Description

The SCOTCH_orderInit routine fills the ordering structure pointed to by ordeptr with all of the data that is passed to it. Thus, all subsequent calls to ordering routines such as SCOTCH_orderCompute, using this ordering structure as parameter, will place ordering results in fields *cblkptr, rangtab, permtab, peritab.

cblkptr is the pointer to a SCOTCH_Num that will receive the number of column blocks produced, rangtab is the array that holds the column block span information, of size vertnbr+1, permtab is the ordering permutation array, of size vertnbr, *and* peritab *is the inverse ordering permutation array, of size* vertnbr (see section 8.1.3).

It should be the first function to be called upon a SCOTCH_Ordering structure. When the ordering structure is no longer of use, call function SCOTCH_orderExit to free its internal structures.

Return values

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

8.8.2 SCOTCH_orderExit

Synopsis

```
void SCOTCH_orderExit (SCOTCH_Ordering *  ordeptr);
```

Description

The SCOTCH_orderExit function frees the contents of a SCOTCH_Ordering structure previously initialized by SCOTCH_orderInit. All following calls to SCOTCH_order routines other than SCOTCH_orderInit, using this structure as parameter, may yield unpredictable results.

8.8.3 SCOTCH_orderSave

Synopsis

```
int SCOTCH_orderSave (const SCOTCH_Ordering *  ordeptr,
                      FILE *                   stream);
```

Description

The SCOTCH_orderSave routine saves the contents of the SCOTCH_Ordering structure pointed to by ordeptr to stream stream, in the Scotch ordering format (see section 6.5).

Return values

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

8.8.4 SCOTCH_orderCheck

Synopsis

```
int SCOTCH_orderCheck (SCOTCH_Ordering *  ordeptr);
```

Description

The SCOTCH_orderCheck routine checks the consistency of the given SCOTCH_Ordering structure pointed to by ordeptr.

Return values

SCOTCH_orderCheck returns 0 if ordering data are coherent, and 1 else.

To be continued...

8.9 Error handling routines

The handling of errors is often complex for complex libraries, when error messages should be issued to help the application programmer in finding the error while not interfering with the application.

To match these two requirements, all of the error messages produced by the routines of the LIBSCOTCH library are issued by using the usr-defineable 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 prevent the user from writing error handlers from scratch, the libscotcherr.a library provides error routines that print error messages on the standard error stream stderr and return control to the application.

8.9.1 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.

8.9.2 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.

8.9.3 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 were generated. This routine has not been designed to be reentrant, as it is only a minor help.

8.10 Calling from Fortran

The routines of the libScotch can be called either from C or Fortran. For any C function named SCOTCH_*typeAction* () that is documented in this manual, there exists a SCOTCHF*TYPEACTION* () Fortran counterpart, in which the separating underscore character has been replaced by an "F". Fortran routines have exactly the same parameters as C functions, except for an added trailing INTEGER argument which holds the return value yielded by the function.

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

For routines that read or write data using a FILE * stream in C, the Fortran counterpart uses an INTEGER parameter which is the numer of the logical unit (that is, the file descriptor) to read from or to write to.

For instance, the SCOTCH_graphInit and SCOTCH_graphLoad routines, described in sections 8.4.1 and 8.4.3, respectively, can be called from Fortran by using the following code.

```
INCLUDE "scotchf.h"
DOUBLEPRECISION GRAPH(SCOTCH_GRAPHDIM)
INTEGER RETVAL
...
CALL SCOTCHFGRAPHINIT (GRAPH, RETVAL)
IF (RETVAL > 0) THEN
...
CALL SCOTCHFGRAPHLOAD (GRAPH, 5, 1, 0, RETVAL)
IF (RETVAL > 0) THEN
...
```

8.11 Compiling and linking

The compilation of C or Fortran routines that make use of routines of the LIBSCOTCH requires that either `scotch.h` or `scotchf.h` be included, respectively.

The routines of the LIBSCOTCH library are grouped in file `libscotch.a`. File `libcommon.a` contains low-level, generic purpose routines that perform sorting, return random integer numbers, compute random permutations, etc. Default error routines that print an error message and exit are provided in library file `libscotcherr.a`.

Therefore, the linking of applications that make use of the LIBSCOTCH library is performed by using the following options: `-lscotch -lcommon -lscotcherr -lm`. If you provide error handling routines by yourself (see section 8.9), do not link with file `libscotcherr.a`.

9 Installation

The SCOTCH distribution is publicly available for research purposes and may be licensed for commercial application. An export distribution, containing the executables for several machines and operating systems, along with technical documentation and sample files, may be obtained via `WWW` from the SCOTCH page at `http://www.labri.u-bordeaux.fr/Equipe/PARADIS/membre/pelegrin/scotch/` .

The extraction process will create a `scotch_3.4` directory, with four subdirectories named "`bin`", "`doc`", "`grf`", and "`tgt`".

- The "`bin`" directory has subdirectories for each architecture for which the distribution is available (for example `mips_sgi_irix6` or `sparc_sun_solaris2`). Every subdirectory contains a whole set of executables and libraries, compiled for the proper architecture.
- The "`doc`" directory contains the PostScript file of this user's guide, as well as other relevant data.
- The "`grf`" directory contains the source and geometry files of some classical test graphs, coded with our format. To avoid wasting disk space and network bandwidth, only a few of them have been included in the distribution.
- The "`tgt`" directory contains target architecture files, either hand-made (for algorithmically-defined architectures) or computed from source graphs with the `amk_src` program. Table 1 lists all target file types present in the "`tgt`" directory.

Should you have any trouble obtaining, installing, or running SCOTCH, feel free to contact the author at the address given on the cover page of this report. Moreover, all SCOTCH users are welcome to send a mail to the author so that they can be added to the SCOTCH mailing list, and be personally informed of new releases and publications.

10 Examples

This section contains chosen examples destined to show how the programs of the SCOTCH project interoperate and can be combined. It is supposed that the current

Name	Architecture	Number of vertices
`ccc`n`.tgt`	CCC graph	$n \times 2^n$
`fft`n`.tgt`	FFT graph	$(n+1) \times 2^n$
`k`n`.tgt`	Complete graph (bus)	n
`m`x`x`y`.tgt`	Bidimensional mesh	$x \times y$
`m`x`x`y`x`z`.tgt`	Tridimensional mesh	$x \times y \times z$
`t`x`x`y`.tgt`	Bidimensional torus	$x \times y$
`t`x`x`y`x`z`.tgt`	Tridimensional torus	$x \times y \times z$
`tl`n`.tgt`	Tree-leaf graph	2^n
`ub2_`n`.tgt`	Binary de Bruijn graph	2^n
`vcmplt.tgt`	Variable-sized complete graph	–
`vhcub.tgt`	Variable-sized hypercube	–
`void.tgt`	No target (I/O test)	1

Table 1: Target architecture files.

directory is directory "`scotch3.4`" of the SCOTCH distribution. Character "%" represents the shell prompt.

- Map a 32 by 32 bidimensional mesh source graph onto a 256-node hypercube, and save the result to file `/tmp/brol.map`.

  ```
  % smk_m2 32 32 | map - tgt/h8.tgt /tmp/brol.map
  ```

- Build the OpenInventor file `graph.iv` that contains the display of a source graph whose source and geometry files are named `graph.src` and `graph.xyz`.

  ```
  % out -Mn -Oi graph.src graph.xyz - graph.iv
  ```

  Although no mapping data is required because of the "`-Mn`" option, note the presence of the dummy input mapping file name "`-`", which is needed to specify the output visualization file name.

- Given the source and geometry files `graph.src` and `graph.xyz` of a source graph, map the graph on a 8 by 8 bidimensional mesh and display the mapping result on a color screen by means of the public-domain `ghostview` PostScript previewer.

  ```
  % map graph.src tgt/m8x8.tgt | out graph.src graph.xyz
  '-Op{c,f,l}' | ghostview -
  ```

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

  ```
  % amk_ccc 3 | acpl - /tmp/ccc3.tgt
  % gunzip -c graph.src.gz | map - /tmp/ccc3.tgt /tmp/brol.map
  ```

  To speed up target architecture loading in the future, the decomposition-defined target architecture is compiled by means of `acpl`.

- Build an architecture graph which is the subgraph of the 8-node de Bruijn graph restricted to vertices labeled 1, 2, 4, 5, 6, map graph `graph.src` onto it, and save the result to file `/tmp/brol.map`.

  ```
  % (smk_ub2 3; echo 5 1 2 4 5 6) | amk_src -L | map graph.src -
  /tmp/brol.map
  ```

  Note how the two input streams of program `amk_src` (that is, the de Bruijn source graph and the five-elements vertex label list) are concatenated into a single stream to be read from the standard input.

- Output the pattern of the adjacency matrix associated with graph `graph.src.gz` to the encapsulated PostScript file `graph.pattern.ps`.

  ```
  % gunzip -c graph.src.gz | out - - - -Gn -Mn '-Om{e}'
  graph.pattern.ps
  ```

- Output the pattern of the factored reordered matrix associated with graph `graph.src.gz` to the encapsulated PostScript file `graph.factor.ps`.

  ```
  % gunzip -c graph.src.gz | ord - -F- /dev/null | out - - - -Gn
  -Mn '-Om{e}' graph.pattern.ps
  ```

- Compile and link the user application `brol.c` with the LIBSCOTCH library, using the default error handler.

  ```
  % cc brol.c -o brol -lscotch -lscotcherr -lm
  ```

  Note that the mathematical library should also be included, after all of the SCOTCH libraries.

Credits

I wish to thank all of the people cited below :

- Patrick Amestoy collaborated to the design of the Halo Approximate Minimum Degree algorithm [42], that has been embedded into SCOTCH 3.3, and gave me versions of his Approximate Minimum Degree algorithm, available since version 3.2, and of his Halo Approximate Minimum Fill algorithm, available since version 3.4;

- Alex Pothen kindly gave me a version of his Multiple Minimum Degree algorithm, that has been embedded into SCOTCH 3.2;

- Luca Scarano, visiting Erasmus student from the *Universitá degli Studi di Bologna*, coded the multi-level algorithm in SCOTCH 3.1;

- David Sherman proofread version 3.2 of this manual.

References

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

[2] C. Ashcraft. Compressed graphs and the minimum degree algorithm. *SIAM J. Sci. Comput.*, 16(6):1404–1411, 1995.

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

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

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

[6] I. Duff. On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Software*, 7(3):315–330, September 1981.

[7] I. S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection. Technical Report TR/PA/92/86, CERFACS, Toulouse, France, October 1992.

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

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

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

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

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

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

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

[15] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. *ACM Trans. Math. Software*, 2:322–330, 1976.

[16] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. TR 94-063, University of Minnesota, 1994. To appear in *IEEE Trans. on Parallel and Distributed Systems*, 1997.

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

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

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

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

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

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

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

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

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

[26] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2(4):225–231, December 1973.

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

[28] G. Karypis and V. Kumar. METIS – unstructured graph partitioning and sparse matrix ordering system – version 2.0. Technical report, University of Minnesota, June 1995.

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

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

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

[32] M. Laguna, T. A. Feo, and H. C. Elrod. A greedy randomized adaptative search procedure for the two-partition problem. *Operations Research*, pages 677–687, July 1994.

[33] C. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, W. Hillis, B. Kuszmaul, M. Pierre, D. Wells, M. Wong, S. Yang, and R. Zak. The network architecture of the Connection Machine CM-5. Technical report, Thinking Machines, juillet 1992.

[34] C. Leiserson and J. Lewis. Orderings for parallel sparse symmetric factorization. In *Third SIAM Conference on Parallel Processing for Scientific Computing, Tromsø*. SIAM, 1987.

[35] M. Lin, R. Tsang, D. H. C. Du, A. E. Klietz, and S. Saroff. Performance evaluation of the CM-5 interconnection network. In *Proceedings of CompCon Spring'93*, 1993.

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

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

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

[39] F. Pellegrini and J. Roman. Experimental Analysis of the Dual Recursive Bipartitioning Algorithm for Static Mapping. Technical Report, LaBRI, Université Bordeaux I, August 1996. Availble at URL `http://www.labri.u-bordeaux.fr/~pelegrin/papers/scotch_expanalysis.ps.gz`.

[40] F. Pellegrini and J. Roman. SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *Proceedings of HPCN'96, Brussels,* LNCS 1067, pages 493–498, April 1996.

[41] F. Pellegrini and J. Roman. Sparse matrix ordering with SCOTCH. In *Proceedings of HPCN'97, Vienna,* LNCS 1225, pages 370–378, April 1997.

[42] F. Pellegrini, J. Roman, and P. Amestoy. Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. In *Proceedings of Irregular'99, San Juan*, April 1999. To appear.

[43] A. Pothen and C.-J. Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Software*, 16(4):303–324, December 1990.

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

[45] E. Rothberg. Performance of panel and block approaches to sparse Cholesky factorization on the iPSC/860 and Paragon multicomputers. In *Proceedings of SHPCC'94, Knoxville*, pages 324–333. IEEE, May 1994.

[46] E. Rothberg and A. Gupta. An efficient block-oriented approach to parallel sparse Cholesky factorization. In *Supercomputing'93 Proceedings*. IEEE, 1993.

[47] E. Rothberg and R. Schreiber. Improved load distribution in parallel sparse Cholesky factorization. In *Supercomputing'94 Proceedings*. IEEE, 1994.

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

[49] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2:135–148, 1991.

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

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