SCOTCH 3.1 **User's Guide***

François Pellegrini
LaBRI, URA CNRS 1304
Université Bordeaux I
351 cours de la Libération, 33405 TALENCE, FRANCE
`pelegrin@labri.u-bordeaux.fr`

Revision 1.0
June 5, 1997

Abstract

The efficient execution of a parallel program on a parallel machine requires good placement of the communicating processes of the program onto the processors of the machine. When both the program and the machine are modeled in terms of weighted unoriented graphs, this problem amounts to static graph mapping.

This document describes the capabilities and operations of SCOTCH, a software package devoted to graph mapping, based on the Dual Recursive Bipartitioning algorithm. Predefined mapping strategies allow for recursive application of any of several graph bipartitioning methods, including Fiduccia-Mattheyses, Gibbs-Poole-Stockmeyer, and multi-level methods. SCOTCH can map any weighted process graph onto any weighted target graph, whether they are connected or not. We give brief descriptions of the algorithm and bipartitioning methods, detail the input/output formats, instructions for use, and installation procedures, and provide a number of examples.

*This work was supported by the French GDR PRS.

Contents

1 Introduction

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 [6]. Therefore, many studies have been carried out in order to find sub-optimal solutions in reasonable time. Specific algorithms have been proposed for mesh [22] and hypercube [4, 9] topologies. 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 [1, 10, 15, 17, 26]. 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 [9].

This document describes the capabilities and operations of Scotch, a software package devoted to graph mapping that is based on the Dual Recursive Bipartitioning algorithm. Scotch allows the user to map efficiently any kind of weighted process graph onto any kind of weighted architecture graph, whether they are connected or not, in an observed running time linear in the number of source edges and logarithmic in the number of target vertices. 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 Dual Recursive Bipartitioning algorithm. Section 4 defines the formats of the files used in Scotch, and section 5 describes the programs of the Scotch distribution. Section 6 explains how to obtain and install the Scotch distribution. Finally, some practical examples are given in section 7.

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 static mapping by the means of graph theory, using a "divide and conquer" approach. This work has resulted in the development of the Dual Recursive Bipartitioning (or DRB) mapping algorithm and the analysis of several graph bipartitioning heuristics, all of which have been implemented and embodied in the SCOTCH software package for static mapping [23, 24].

2.2 Availability

The programs developed within the SCOTCH project, including the mapping, 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 6 describes how to obtain it via `WWW` or anonymous `ftp`.

3 Static mapping by Dual Recursive Bipartitioning

For a detailed description of the mapping algorithm and an extensive analysis of its performance, please refer to [23, 25]. In the next sections, we will only outline the most important aspects of the algorithm.

3.1 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, $\ldots$, 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.

3.2 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 [4, 9, 13], 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 [9] on the CM-2, and by Hendrickson and Leland [14] 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

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

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

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.

3.3 The Dual Recursive Bipartitioning algorithm

Our mapping algorithm uses a *divide and conquer* approach to recursively allocate subsets of processes to subsets of processors [23]. 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.

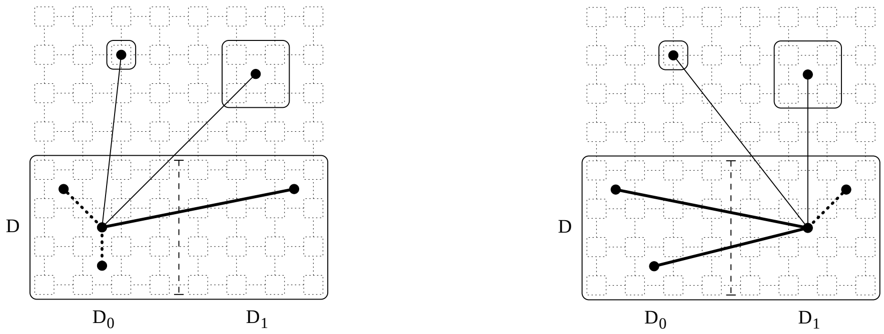
All these routines are seen as black-boxes by the mapping program, which can thus accept any kind of target architecture and process bipartitioning functions.

3.4 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.



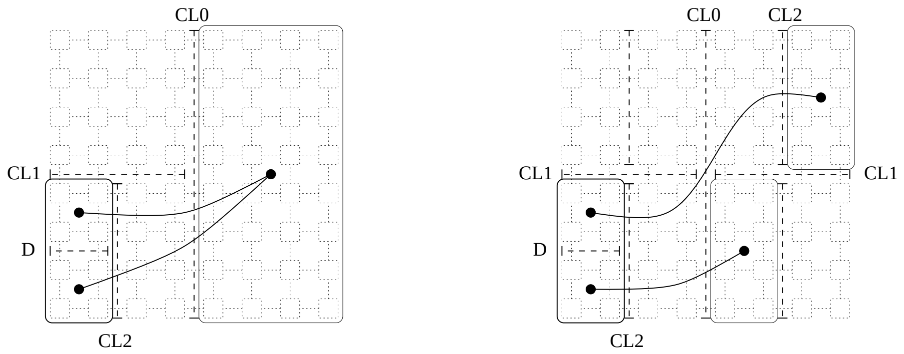
a. Initial position. **b.** After one vertex is moved.

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

3.5 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 of depth-first



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").

and breadth-first sequencing schemes with respect to the structure of the source and target graphs is discussed in [25].

3.6 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.

Backtracking

The backtracking method explores all possible solutions, by successively considering the assignment of every vertex to each subdomain. It always finds an optimal solution, but its running time is exponential in the number of source graph vertices. Therefore, its use should be restricted to bipartitioning jobs of small size, when local optimization becomes significant and worthwhile. However, even in this case, the greedy-graph-growing method should be preferred.

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 [5] is an almost-linear improvement of the famous Kernighan-Lin algorithm [18]. 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 [25].

As several authors quoted before [12, 16], 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 [8]. 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 [7], and by Simon in [27].

Greedy-graph-growing

This greedy algorithm, which has been proposed by Karypis and Kumar [15], belongs to the GRASP ("*Greedy Randomized Adaptative Search Procedure*") class [19]. 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 [1, 15, 11] 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

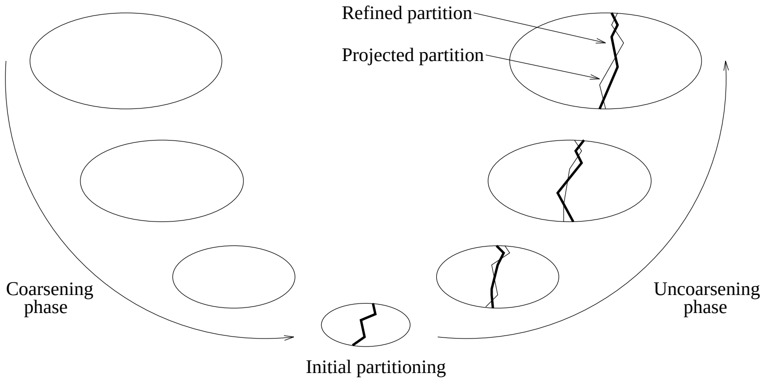


Figure 3: The multi-level partitioning process. In the uncoarsening phase, the light and bold lines represent for each level the projected partition obtained from the coarser graph, and the partition obtained after refinement, respectively.

in conjunction 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.

Random

Two random methods have been actually implemented. The first one computes a purely random bipartition, accounting neither for load balance nor for communication minimization, and is used to obtain reference values (average edge dilation and average vertex load) which can help in evaluating other methods. The second method selects at random the part to which a given vertex is assigned only if both parts can host the vertex without causing load imbalance; otherwise, the vertex is directly assigned to the only part it can fit in. This method has been designed to provide random initial partitions to iterative methods for which initial balance is mandatory.

4 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, **numberings always start from zero**, not one.

4.1 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 first line of a 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 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: Source file representing a cube.

The Scotch distribution contains a file converter, `scv`, which automatically translates several graph formats into source graph format. It is described in section 5.2.8.

4.2 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 5 shows the contents of the 3D geometry file associated with the source graph of figure 4.

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

Figure 5: Geometry file associated with the source file of figure 4.

4.3 Target files

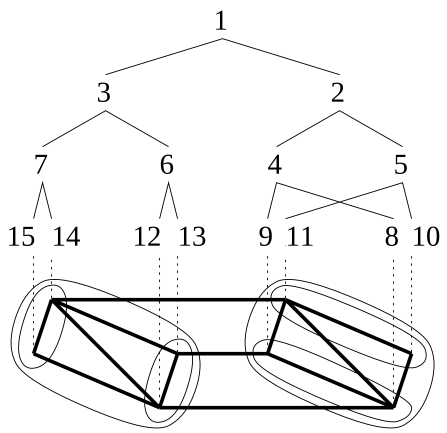
Target files describe the architectures onto which source graphs are mapped. Instead of containing the structure of the target graph itself, as source files do, target files define how target graphs are bipartitioned and give the distances between all pairs of vertices (that is, processors). Keeping 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.

4.3.1 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 6 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 6: Target decomposition file for UB$(2,3)$. The terminal numbers associated with every processor define a unique recursive bipartitioning of the target graph.

4.3.2 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*

Defines a tree-leaf architecture with *height* levels and $2^{height} - 1$ 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 7. Only the leaves are used to map processes, but distances between them are computed by considering the whole tree. This graph is used to represent multi-stage machines with constant bandwidth, such as the CM-5 [20] for which experiments have shown that bandwidth is constant between every pair of processors and hardly depends on network congestion [21], or the SP-2 with power-of-two number of nodes.

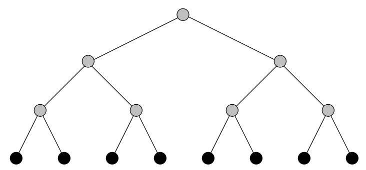


Figure 7: The "tree-leaf" graph of height 3. Processors are drawn in black and routers in grey.

4.4 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 8 shows the result obtained when mapping the source graph of figure 4 onto the target architecture of figure 6. This one-to-one embedding of H(3) into $\mathrm{UB}(2,3)$ has dilation 1, except for one hypercube edge which has dilation 3.

4.5 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 9 shows the list made from three vertices of labels 2, 45, and 7.

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

Figure 8: Mapping file obtained when mapping the hypercube source graph of figure 4 onto the binary de Bruijn architecture of figure 6.

```
3  2  45  7
```

Figure 9: Example of vertex list with three vertices of labels 2, 45, and 7.

5 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 mapper itself.
- Mapping handling programs, which are the mapping performance analyzer and the graph and mapping visualization program.

The general architecture of the Scotch project is given in figure 10.

5.1 Invocation

The programs comprising the Scotch project have been designed to run in command-line mode without any interactive prompting, so that they can be called easily from other programs by means of "`system ()`" or "`popen ()`" system calls, or be piped together on a single shell command line. In order to facilitate this, whenever a stream name is asked for (either on input or output), the user may put a single "`-`" to indicate standard input or output. Moreover, programs read their input in the same order as stream names are given in the command line. It allows them to read all their data from a single stream (usually the standard input), provided that these data are ordered properly.

A brief on-line help is provided with all the programs. To get this help, use the "`-h`" option after the program name. The case of option letters is not significant, except when both the lower and upper cases of a letter have different meanings. When passing parameters to the programs,

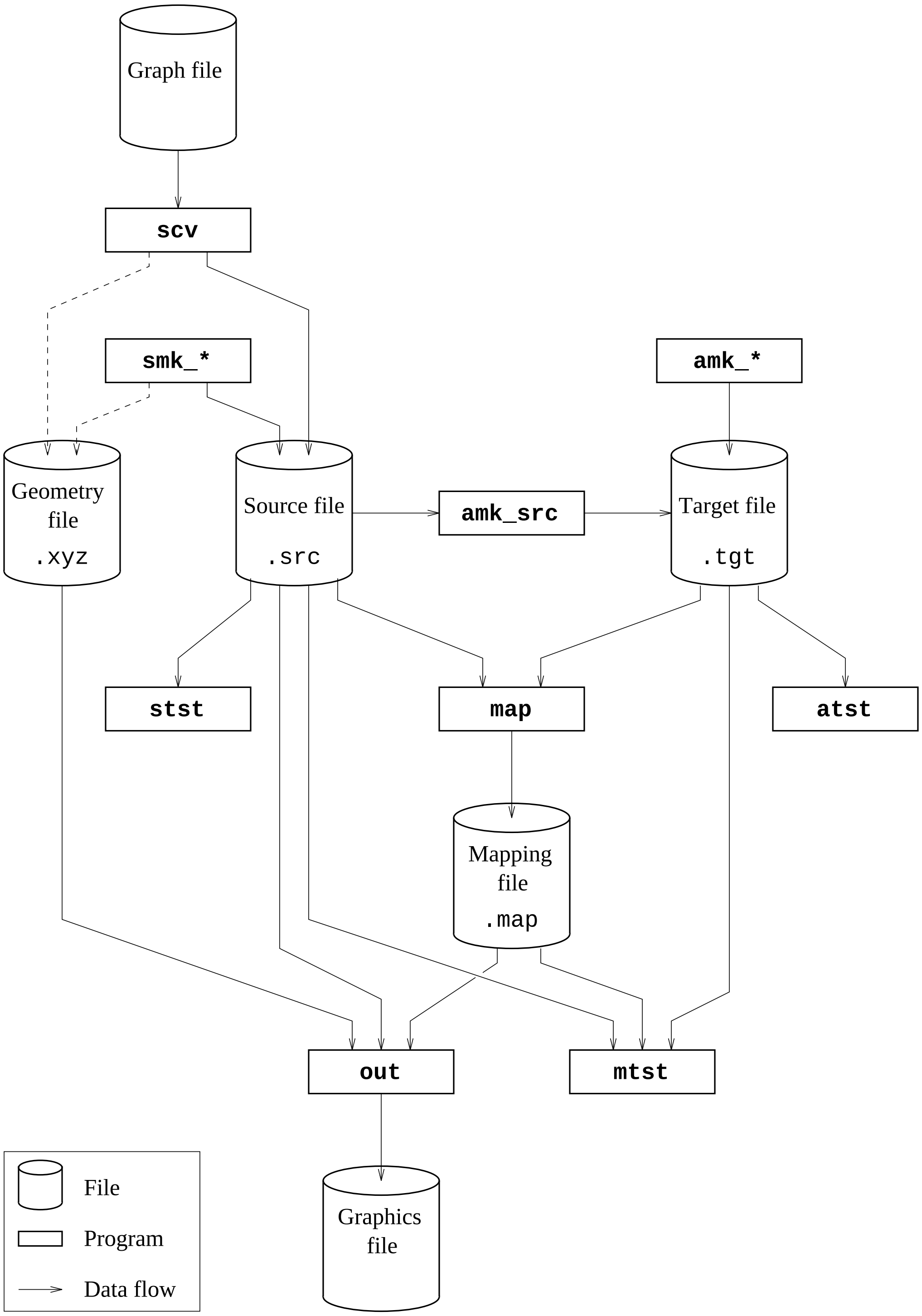


Figure 10: General architecture of the Scotch project.

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 7.

Error messages are standardized, but may not be fully explanatory. However, most of the errors you may run into should be related to file formats, and located in "`...Load()`" routines. In this case, compare your data formats with the definitions given in section 4.

5.2 Description

5.2.1 `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 4.3.1 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 a lot of time when many mappings are to be performed onto the same target architecture.

Options

`-h` Display the program synopsis.

5.2.2 `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 `-m` option selects the bipartitioning strategy used to build the recursive bipartitioning of the source graph. For regular, unweighted, topologies, the `-mgfx` should work best. For irregular or weighted graphs, use the default strategy, which is more flexible.

Options

`-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.

`-m`*strategy*
: Use mapping strategy *strategy* to map the architecture graph onto the complete graph. See the manual page of `map` for the syntax of the strategy string.

5.2.3 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*

`amk_ub2` *dim* [*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.

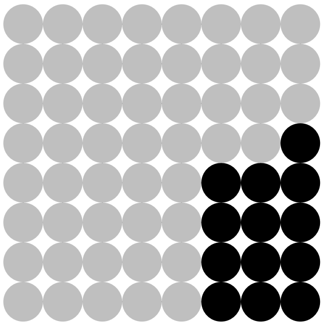
Program `amk_p2` outputs the target architecture file of a weighted path graph with two vertices, the weights of which are given as parameters.
This simple target topology is used to bipartition a source graph into two weighted parts with as few cut edges as possible. In particular, it is used to compute 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 11).

Program `amk_ub2` outputs the target architecture file of a binary de Bruijn graph of dimension *dim*. Vertices are labeled according to the decimal value of their binary representation.
The algorithmic bipartitioning method implemented in this program leads to decompositions which are less efficient than the ones computed by `amk_src` from source de Bruijn graphs generated with `smk_ub2`. This program should therefore be considered obsolete, except for research purposes.

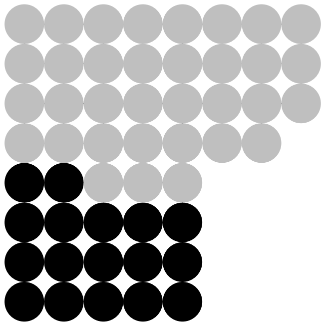
Options

`-h` Display the program synopsis.

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



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

`n` Nested dissection.

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

5.2.4 `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.

5.2.5 `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 the Dual Recursive Bipartitioning approach 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 bipartitioning 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. Bipartitioning methods are used to define bipartitioning strategies by means of

selection, combination, grouping, and condition operators.
Available methods include a multi-level algorithm that uses other 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 bipartitioning strategy that is used for mapping.

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

`-b`*val*

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

`-e`*exe*

Select execution scheme *exe*. The execution scheme is the order in which the tree of recursive bipartitioning jobs is traversed. It is controlled by several parameters, which can be set individually, or globally by means of composite flags. The currently defined composite flags are listed below.

- `a` Adaptative (composite: -EjSmtpt).
- `b` Breadth-first (composite: -EjSmtpu).
- `d` Depth-first (composite: -Ejlmtpt).

The individual parameters that control the execution scheme are set by the following options flags.

`j`*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.

`m`*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.

`p`*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.

`-h` Display the program synopsis.

`-m`*strategy*

Apply bipartitioning strategy *strategy*. 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 bipartitioning methods are listed below.

`b` Backtracking method.

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

`move=`*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.

`pass=`*nbr*

Set the maximum number of optimization passes performed by the algorithm. Value -1 stands for an infinite number of passes, that is as many as needed by the algorithm to converge.

`type=`*type*

Selects the type of algorithm. *type* is `c` if the complete Fiduccia-Mattheyses algorithm is to be used, and `f` if only the faster version is needed. The faster version differs from the complete one by the lack of the hill-climbing feature; it is only a gradient-like method, which it is mainly intended to refine partitions in the uncoarsening phase of the multi-level method.

`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 can not be less that 0.5 (case of a perfect matching), and can not 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.

`r` Random method.

`type=`*type*

Set the type of random method used. *type* is `b` for the balanced random method, and `u` for the unbalanced random method.

`x` Exactifying method.

`-o`*output_source_file*

Output source file, sorted by decreasing vertex weights, to *output_source_file*. Since weighted source graphs are always sorted once they have been read, this option makes it possible to get back the resulting sorted graph, and therefore to speed-up future loads. When graphs are already sorted or graph vertices all have the same weight, no sorting is performed on loading.

`-s`*obj*

Force unity weights on source graph vertices and/or edges. This option "unweights" weighted source graphs, which can be saved on their unweighted form with option `-o`. *obj* may contain several of the following switches.

`e` Unweight edges.

`v` Unweight vertices.

`-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`.

`c` Communication information.

`m` Mapping information.

`t` Timing information.

Default option set is "`-B0.005 -Ea -Mm{low=(gf|h),asc=f{move=b}}`".

5.2.6 `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, which begin in "M", deal with process mapping statistics, while the following lines, which begin in "J", deal with communication statistics. The first mapping line gives the number of processors used by the mapping, followed by the usage ratio with respect to the total number of processors available in the architecture. The second mapping line gives the minimum, maximum, average, and average distance to the mean of the mapping. The first communication line gives the average dilation of the edges, followed by the sum of all edge dilations. The second communication line gives the average expansion of the edges, followed by the value of function f_C. The third communication line gives the average cut of the edges, followed by the number of cut edges. The fourth 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.

5.2.7 `out`

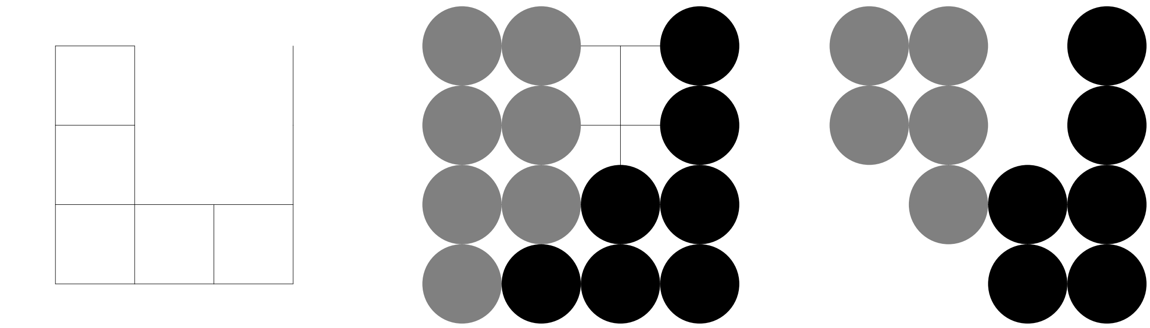
Synopsis

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

Description

The `out` program is the 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 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.
The number of mapping pairs contained in the input mapping file may be different 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 12.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 12.c shows how the display of the same mapping can be restricted to a subgraph of the original graph.



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

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

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

Figure 12: 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.

Options

`-g`*parameters*
: Geometry parameters.

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

`-h` Display the program synopsis.

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

`-o`*format*[{*parameters*}]
: Specify the type of output visualization file, with optional parameters within curly braces and separated by commas.

  - `i` 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`.

- `p` 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`.
  - `d` Display the mapping disks. Opposite of `a`.
  - `e` Encapsulated PostScript output, suitable for LaTeX use with `epsf`.
  - `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 `s`.
  - `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 "`-G2 -Op{c,d,f,s,v,x=0.0,X=1.0,y=0.0,Y=1.0}`".

5.2.8 `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 our source graph format, along with its associated geometry file whenever geometrical data is available. At the time being, it only accepts four input formats: the Chaco graph format [12], S. W. Hammond's format, H. D. Simon's simple format, and the Harwell-Boeing collection format [3].

Options

- `-h` Display the program synopsis.
- `-i`*format*
  Specify the type of input graph to be converted into our source graph format.

`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` Horst Simon's simple graph format.

5.2.9 `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 7, 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).

5.2.10 stst

Synopsis

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

Description

The program `stst` is the source graph tester. It gives some statistics on source graphs, regarding edge weights, vertex weights, and vertex degrees.

Options

`-h` Display the program synopsis.

6 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/~pelegrin/scotch/`, or by anonymous `ftp` at `ftp.u-bordeaux.fr` in directory `/pub/Local/Info/Software/Scotch`. It is labeled `scotch_3.1A.tar.gz`, and is in compressed (`gzip`) archive (`tar`) form.

The extraction process will create a `scotch3.1` directory, with four subdirectories named "`bin`", "`doc`", "`src`", and "`tgt`".

- The "`bin`" directory has subdirectories for each architecture for which the distribution is available (for example `mips_sgi_irix4` or `sparc_sun_solaris2`). Every subdirectory contains a whole set of executables, compiled for the proper architecture.

- The "`doc`" directory contains the PostScript file of this user's guide, as well as other relevant data.

- The "`src`" 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. Others can be directly uploaded from the Scotch graph collection, which can be accessed from the Scotch `WWW` page, or by anonymous `ftp` at `ftp.u-bordeaux.fr` in directory `/pub/Local/Info/Software/Scotch/Graphs`.

  Graphs prefixed with "`ar_`" are classical regular graphs, computed by programs of the `smk_*` family.
  Graphs prefixed with "`fe_`" are unstructured meshes related to fluid dynamics, structural mechanics, or combinatorial optimization problems. They have been collected from public distributions [3] or from individuals (S. Hammond and H. Simon).
  Graphs prefixed with "`nd_`" are valuated interprocess communication graphs issued from a parallel implementation of a sparse block Cholesky factorization solver, and represent partitions of the unknowns induced by a nested dissection method. The most imbalanced graphs are the direct output of the nested dissection process. Others are obtained from these by

Name	Architecture	Number of vertices
ccc$n$.tgt	CCC graph	$n \times 2^n$
fft$n$.tgt	FFT graph	$(n+1) \times 2^n$
kn.tgt	Complete graph (bus)	n
mxxy.tgt	Bidimensional mesh	$x \times y$
mxxyxz.tgt	Tridimensional mesh	$x \times y \times z$
txxy.tgt	Bidimensional torus	$x \times y$
txxyxz.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
void.tgt	No target (I/O test)	1

Table 1: Target architecture files.

means of a refinement process, in which heavier vertices are split into cliques of lighter vertices. This allows for better granularity of the problems, at the expense of vertex and (mostly) edge creations. Therefore, most of these graphs have very high degree. See [2] for reference.

- 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.

7 Examples

This section contains chosen examples destined to show how the programs of the Scotch project interoperate and can be combined. It is supposed that the current directory is directory "`scotch3.1`" 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 -M '-Oi' graph.src graph.xyz - graph.iv
```

Although no mapping data is required because of the "-M" 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 16 by 16 bidimensional mesh and display the mapping result on a color screen by means of the public-domain `ghostview` PostScript previewer.

```
% map graph.src tgt/m16x16.tgt | out graph.src graph.xyz '-Op{c,f,l}' |
ghostview -
```

- Build a 16-node de Bruijn 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_ub2 4 | acpl - /tmp/ub2_4.tgt
% gunzip -c graph.src.gz | map - /tmp/ub2_4.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.

Acknowledgements

I wish to thank Luca Scarano, visiting Erasmus student from the Universitá degli Studi di Bologna, who coded the multi-level algorithm. Many thanks also to Steve Hammond, Bruce Hendrickson, and Horst Simon, who kindly gave me the test graphs that they had, and to David Sherman, who proofread this manual.

References

[1] 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.

[2] P. Charrier and J. Roman. Partitioning and mapping for parallel nested dissection on distributed memory architectures. In *LNCS 634 – Proceedings of CONPAR'92*, pages 295–306. Springer-Verlag, September 1992.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

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

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

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

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

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

[14] 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.

[15] 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.

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

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

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

[19] 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.

[20] 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, July 1992.

[21] 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.

[22] D. M. Nicol. Rectilinear partitioning of irregular data parallel computations. *Journal of Parallel and Distributed Computing*, 23:119–134, 1994.

[23] 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.

[24] F. Pellegrini. *Application de méthodes de partition à la résolution de problèmes de graphes issus du parallélisme*. Thèse de Doctorat, LaBRI, Université Bordeaux I, 351 cours de la Libération, 33405 Talence, France, January 1995.

[25] 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`.

[26] 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.

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