SCOTCH 3.0 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 1.2
November 23, 1995

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 and Gibbs-Poole-Stockmeyer 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.

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

*This work was supported by the French GDR PRS.

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, the problem is called *mapping*. The mapping problem 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 processors as close to each other as possible.

A mapping is called *static* if it is computed prior to the execution of the program and is never modified, and *dynamic* when processes are moved between processors 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 [19] 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, 13, 15, 22, 24]. 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, 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.
The rest of this paper 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 (parallel ALgorithmics and ENvironments) 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 `map` mapping program [20, 21].

2.2 Availability

The programs developed within the SCOTCH project, including the mapping, visualization, and graph building programs, are copyrighted by 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 [20, 21]. 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*, the vertices of which represent the processors of the target architecture, and the edges of which the communication links between processors. Vertex- and edge- valuations associate with every vertex v_T and every edge e_T of T integer numbers $w_T(v_T)$ and $w_T(e_T)$ which estimate the computational power of the corresponding processor and the cost of traversal of the 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 is mapped onto processor v_T, and $\rho_{S,T}(e_S) = \{e_T^1, e_T^2, \ldots, e_T^n\}$ if communication channel e_S is routed through communication links e_T^1, e_T^2, ..., e_T^n. $|\rho_{S,T}(e_S)|$ denotes the dilation of edge e_S, *i.e.* the number of edges of $E(T)$ used to route e_S.

3.2 Cost function

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:

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

This function, which has already been considered by several authors for hypercube target topologies [4, 9, 11], has several interesting properties:

- it is simple, easy to compute, and allows incremental updates performed by iterative algorithms;
- its minimization favors the mapping of intensively intercommunicating processes onto nearby processors;
- it models the traffic on the interconnection network and thus the risk of congestion, regardless of the type of routing implemented on the target machine (store-and-forward or cut-through).

The strong positive correlation between values of this function and effective execution times has been experimentally verified by Hammond [9, page 48] on the CM-2, and by Hendrickson and Leland [12, page 683] on the nCUBE 2.

3.3 The Dual Recursive Bipartitioning algorithm

Our mapping algorithm is based on a *divide and conquer* approach. It proceeds by recursive allocation of subsets of processes to subsets of processors, until either the subsets of processors are restricted to one element or the subsets of processes are empty. At each step, the algorithm bipartitions a subset of processors, also called a *domain*, into two disjoint subdomains, and calls a *graph bipartitioning algorithm* to map the associated subset of processes onto the two subdomains (see figure 1), 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.

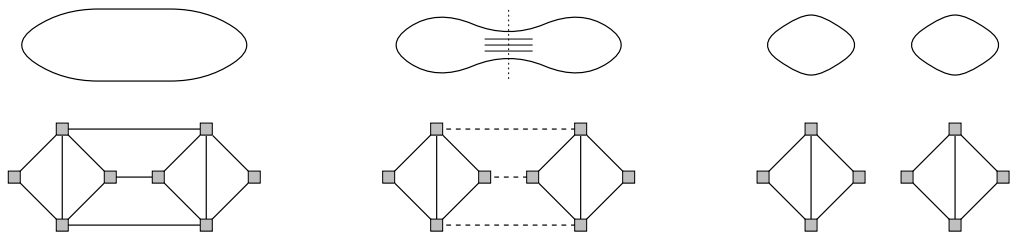


Figure 1: Dual bipartitioning of architecture and process graphs, which divides a mapping problem into two smaller subproblems.

This algorithm lies on the ability to define five main objects:

- a *domain structure*, which represents a set of processors of 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 in the subgraph bipartitioning algorithms to compute the communication function to minimize, since it allows us to estimate the dilation of edges linking 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 *subgraph structure*, which represents the subgraph induced by a subset of the vertex set of the original source graph.

- A *subgraph bipartitioning function*, which bipartitions subgraphs in two disjoint pieces to be mapped onto the two subdomains computed by the domain bipartitioning function.

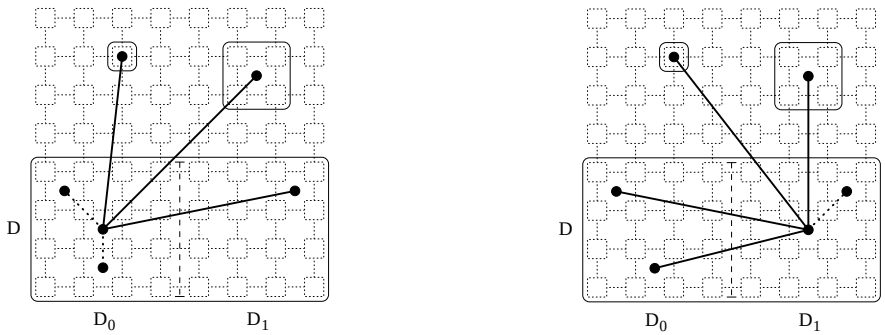
All these routines are seen as black-boxes by the mapping program, which can thus accept any kind of target architecture and process bipartitioning functions.

3.4 Partial cost function

The production of efficient complete mappings requires that all graph bipartitionings enforce 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

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

which accounts for the dilation of edges internal to subgraph S' as well as for the edges which belong to the cocycle of S', as shown in figure 2.
Taking into account partial mappings issued by previous bipartitionings makes it possible to avoid local choices that might prove globally bad.



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

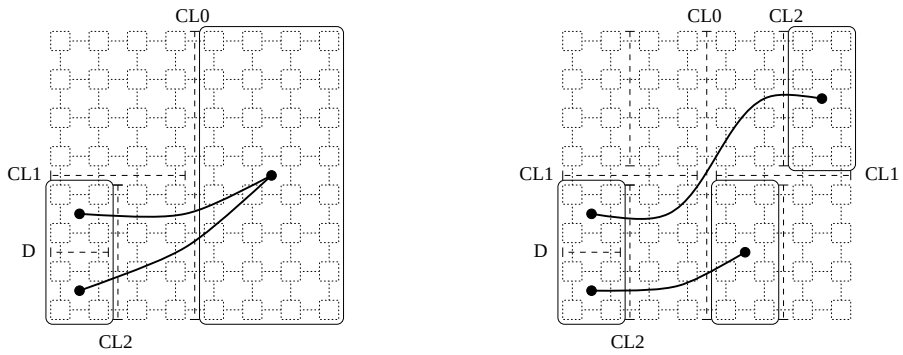
Figure 2: Edges accounted for in the partial communication cost function when bipartitioning the subgraph associated with domain D between the two subdomains D_0 and D_1 of D. Dotted edges are of dilation zero, their two ends being mapped onto the same subdomain. Thin edges are cocycle edges.

3.5 Execution scheme

From an algorithmic point of view, our mapper behaves as a greedy algorithm (the mapping of a process to a subdomain is never reconsidered), 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 called in the left branches have no information on the distance to vertices to be processed by the right branches. On the other hand, sequencing the jobs according to a by-level (breadth-first) traversal of the tree lets a given bipartitioning job have information on the subdomains to which all the processes were allocated during 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 the neighboring 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 cutset 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 thus be more likely to keep these vertices close to each other, as illustrated in figure 3.



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

Figure 3: 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 for vertices in the right branches of the bipartitioning tree are more accurate (C.L. stands for "Cut Level").

3.6 Graph bipartitioning methods

The core of our recursive mapping algorithm uses process graph bipartitioning methods as black boxes. It allows us 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 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 both subdomains. It always finds an optimal solution, but its running time is exponential with respect to the number of vertices. Therefore, its use should be restricted to the leaves of the bipartitioning tree, and only then if the jobs have few vertices.

Exactifier

The goal of this greedy algorithm is to reduce load imbalance 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 the 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 essential, for example when distributing the data of a SPMD program to be run on a SIMD machine.

Fiduccia-Mattheyses

This method is based on the algorithm proposed by Fiduccia and Mattheyses [5], which is itself an improvement of the famous Kernighan-Lin algorithm [16]. It proceeds by greedy iterative refinement of an initial partition, moving vertices between sets so as to minimize the communication cost function of the current partition. The algorithm can accept moves that will increase the cost function, provided that further moves will reduce it; this feature allows hill-climbing from local minima of the cost function.

As several authors quoted before [11, 14], 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.

Gibbs-Poole-Stockmeyer

This bipartitioning method is inspired from an algorithm proposed by Gibbs, Poole, and Stockmeyer to minimize the dilation of graph numberings, that is the maximum absolute value of the difference between the numbers of neighbor vertices [8].

The first phase of the algorithm amounts to cutting the graph into layers. Starting from a randomly chosen vertex which is put into layer 0, layers of increasing numbers are built by putting into layer i all of the as yet unselected neighbors of the vertices of layer $i-1$; this amounts to building a breadth-first spanning tree rooted at the randomly chosen node. When all of the graph vertices have been selected, a vertex of smallest degree belonging to the last layer is randomly chosen as the new root vertex to be put into layer 0, and the layering process is iterated as long as the number of layers increases. Then, starting from layer 0, layer after layer, vertices are assigned to the first subdomain, until half of the total weight has been processed. The remaining vertices are then allocated to the second subdomain.

As for the original Gibbs, Poole, and Stockmeyer algorithm, it is assumed that by maximizing the number of layers, the cardinal –and therefore the cocycle– of the layers is minimized.

This property has already been used by George and Liu to renumber linear systems using the nested dissection method [7], and by Simon in [23].

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. It 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 assigned only to the 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.

Whenever numbers are used to label objects, **numberings always start from zero**, not one.

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.

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.

The SCOTCH distribution contains a file converter, `scv`, which automatically translates several graph formats into source graph format. It is described in section 5.2.7.

```
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.

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 "`2`" if the file contains 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 two or three real numbers which are the (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 the label of which 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 (*i.e.* 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.

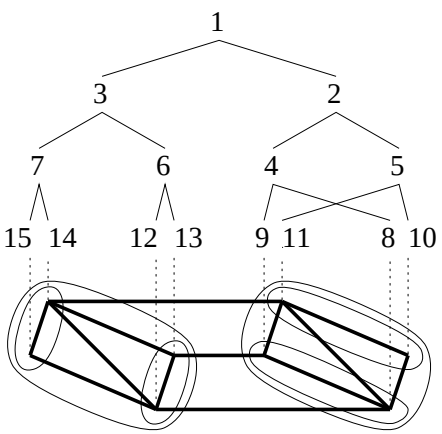
4.3.1 Decomposition-defined architecture files

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 example, 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.

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.




```
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.

`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 [17] for which experiments have shown that bandwidth is constant between every pair of processors and hardly depends on network congestion [18], or the SP-2 with power-of-two number of nodes.

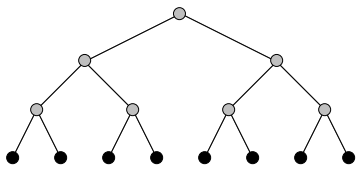


Figure 7: The "tree-leaf" graph of height 3. Processors are drawn in black and routers in grey.

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 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.

Since these are research programs, error messages may not be fully explanatory. However, most of the errors you may run into should be related to file formats. In this case, compare your data formats with the definitions given in section 4.

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 7.

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 list of all the implemented mapping strategies.

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 $l \times 2^{dim} + m$.

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).

`n` Nested dissection.

`o` 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 (*i.e.* 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 program `map` 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. Available methods include an improved implementation of the Fiduccia-Mattheyses algorithm designed to handle valuated graphs, a greedy method derived from the Gibbs, Poole, and Stockmeyer algorithm, and a greedy "exactifying" refinement algorithm designed to balance vertex loads as much as possible. Random and backtracking methods are also provided.
Bipartitioning methods are used to define bipartitioning strategies, which combine them so that the output of a method serves as initial condition for the next. The active strategy is selected with the `-m` option.

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.

`-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.

`-j`*input_job_file*

Load job data from *input_job_file*. This option is used by mapping daemons to share work and results from previous partial mappings computed in parallel. Since running the mapper in parallel is still research, job file formats are not available. Consequently, this option must not be used.

`-m`*strategy*[{*parameters*}]

Apply bipartitioning strategy *strategy* with optional parameters *parameters*. Currently defined strategies are listed below.

`f` Plain Fiduccia-Mattheyses.

`fb` Fiduccia-Mattheyses, followed by a backtracking phase.

`fbx`

Fiduccia-Mattheyses, followed by a backtracking phase, and an exactifying phase.

`fx` Fiduccia-Mattheyses, followed by an exactifying phase.

`g` Plain Gibbs-Poole-Stockmeyer.

`gf` Gibbs-Poole-Stockmeyer, then Fiduccia-Mattheyses.

`gfx`

Gibbs-Poole-Stockmeyer, then Fiduccia-Mattheyses, then the exactifier.

`r` Plain random.

`rf` Random, then Fiduccia-Mattheyses.

`rfx`

Random, then Fiduccia-Mattheyses, then the exactifier.

The different parameters of the `-m` option are listed below. Parameters must be listed between curly braces, and separated by commas.

`l=`*val*

For strategies that use the Fiduccia-Mattheyses method, set the load imbalance tolerance to *val*, which is a floating-point ratio expressed with respect to the ideal load of the partitions.

`r=`*choice*

For strategies that use the random method, select the balanced (`r=b`) or unbalanced (`r=u`) random algorithms.

`s=`*val*

For strategies that use the backtracking method, set the maximum cardinality of the bipartitioning jobs that will be effectively processed; bigger jobs will not be considered, as it would require too much time. This parameter allow strategies to use backtracking only at the last levels of the bipartitioning tree, when local optimization becomes significant and worthwhile.

`x=`*choice*

For strategies that use the Fiduccia-Mattheyses method, set (`x=y`) or unset (`x=n`) the global swap optimization feature.

`-o`*output_source_file*

Output source file, sorted by decreasing vertex weights, to *output_source_file*. Since valuated 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`. Valid values for *obj* are listed below.

`e` Edges.

`v` Vertices.

`-V` Print the program version and copyright.

`-v`*verb*

Set verbose mode to *verb*.

`c` Communication information.

`m` Mapping information.

`t` Timing information.

Default option set is "`-Ea -Mgfx{b=0.005,x=y}`".

5.2.6 `out`

Synopsis

`out` [*input_source_file* [*input_geometry_file* [*input_mapping_file* [*output_picture_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 displayable results. At the time being, `out` only generates plain and encapsulated PostScript files. Therefore, although tridimensional objects can be displayed by means of isometric projection, the tridimensional visualization of mappings is not efficient.

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.

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.

`-Op`[{*parameters*}]

PostScript output selection, with optional parameters within curly braces and separated by commas.

`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}`".

5.2.7 `scv`

Synopsis

`scv` [*input_graph_file* [*output_source_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. At the time being, it only accepts four input formats: the Chaco graph format [11], 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 format.

`h` Steve Hammond's graph format.

`s` Horst Simon's simple graph format.

5.2.8 `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).

5.2.9 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_3.0A.tar.gz, and is in compressed (gzip) archive (tar) form.

The extraction process will create a scotch3.0 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.

  Graphs prefixed with "ar_" are classical regular graphs, computed by programs of the smk_* family.
  Graphs prefixed with "ef_" 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 "de_" 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

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
`void.tgt`	No target (I/O test)	1

Table 1: Target architecture files.

are the direct output of the nested dissection process. Others are obtained from these by 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 "`scotch3.0`" 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
```

- 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` (*i.e.* 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 Steve Hammond and Horst Simon, who kindly gave me the test graphs that they had. Many thanks also 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, sep 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, oct 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, feb 1992.

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

[11] B. Hendrickson and R. Leland. The Chaco user's guide. Technical Report SAND93–2339, Sandia National Laboratories, nov 1993.

[12] 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.

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

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

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

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

[17] 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, jul 1992.

[18] 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.

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

[20] 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.

[21] 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, jan 1995.

[22] 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, jul 1990.

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

[24] O. Zone, D. Vanderstraeten, and R. Keunings. Un solveur direct parallèle basé sur une décomposition de domaine appliqué aux problèmes d'éléments finis. In *Actes des 6èmes Rencontres Francophones du Parallélisme, RenPar'6, Lyon*, pages 113–118. ENS Lyon, jun 1994.

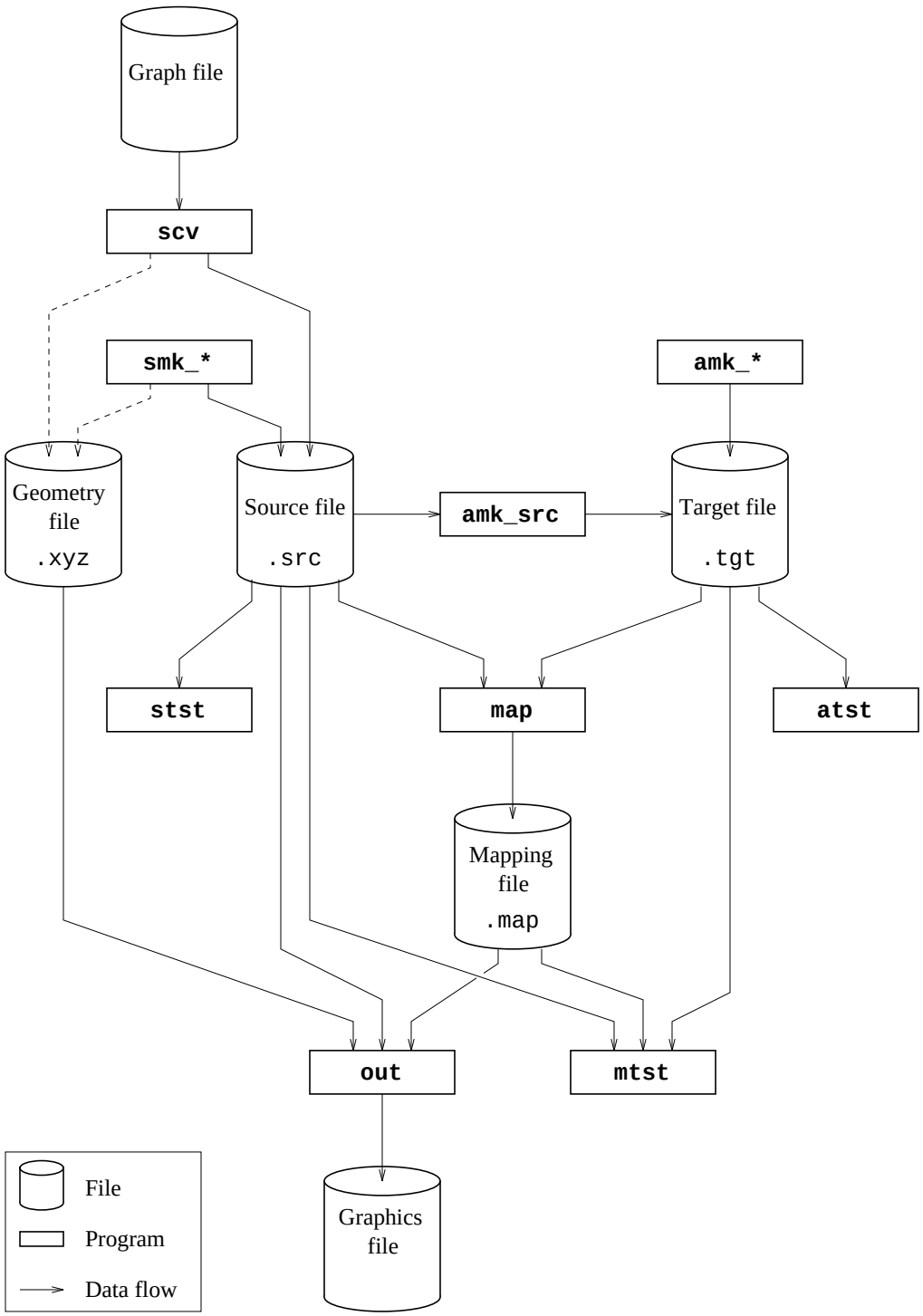
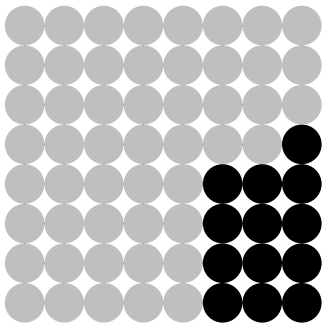
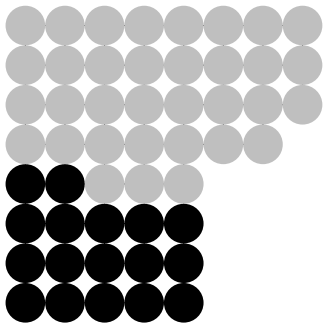


Figure 10: General architecture of the SCOTCH project.

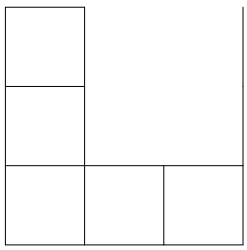


a. Construction of a partition with 13 vertices (in black) on a 8×8 bidimensional mesh architecture.

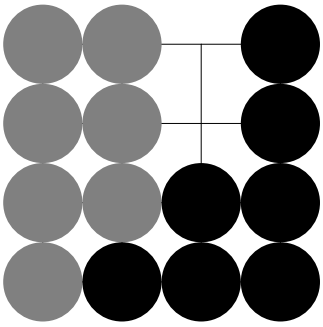


b. Construction of a partition with 17 vertices (in black) on the remaining architecture.

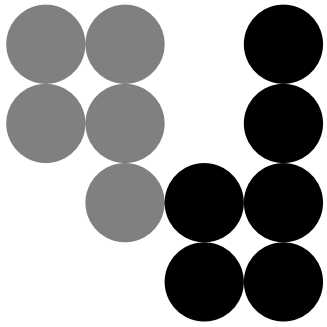
Figure 11: Construction of partitions on a bidimensional 8×8 mesh architecture by weighted bipartitioning.



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: 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.