View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Distillating knowledge about Scotch

François Pellegrini[1]

ENSEIRB-MATMECA, LaBRI and INRIA Bordeaux – Sud-Ouest
351, cours de la Libération, 33405 TALENCE, FRANCE
pelegrin@labri.fr

Abstract. Scotch is a software package for sequential and parallel graph partitioning, static mapping and sparse matrix ordering, and for sequential mesh/hypergraph ordering. It has been designed in a highly modular way, so that new methods can be easily added to it, in order for it to be used as a testbed for the design of new partitioning and ordering methods.
This paper discusses the internal structure of the libScotch library and describes, step by step, how a new method, for instance a sequential vertex separation method, can be added to it.

Keywords. graph partitioning, data structures, API, library, modular programming

1 Introduction

Scotch is a project carried out at the *Laboratoire Bordelais de Recherche en Informatique* (LaBRI) of the Université Bordeaux I, and now within the Bacchus project of INRIA Bordeaux Sud-Ouest. Its goal is to study the applications of graph theory to several problems in scientific computing.

It focused first on static mapping, and has resulted in the development of the Dual Recursive Bipartitioning (or DRB) mapping algorithm and in the study of several graph bipartitioning heuristics [1], all of which have been implemented in the Scotch software package [2]. Then, it focused on the computation of high-quality vertex separators for the ordering of sparse matrices by nested dissection, by extending the work which had been done on graph partitioning in the context of static mapping [3, 4]. Afterwards, the ordering capabilities of Scotch have been extended to native mesh structures, thanks to hypergraph partitioning algorithms. New graph partitioning methods have also been added [5, 6]. Version 5.0 of Scotch was the first to comprise parallel graph ordering routines [7]. Parallel graph partitioning was introduced in version 5.1 [8], and parallel static mapping in version 5.2.

Scotch is available under a dual licensing basis. On the one hand, it is downloadable from the Scotch web page as free/libre software, to all interested parties willing to use it as a library or to contribute to it as a testbed for new partitioning and ordering methods. On the other hand, it can also be distributed, under other types of licenses and conditions, to parties willing to embed

it tightly into closed, proprietary software. The free/libre software license under which SCOTCH 5 is distributed is the CeCILL-C license [9], which has basically the same features as the GNU LGPL ("*Lesser General Public License*") [10]: ability to link the code as a library to any free/libre or even proprietary software, ability to modify the code and to redistribute these modifications.

This paper does not describe how to use the SCOTCH library, which is already documented in the SCOTCH and PT-SCOTCH user's manuals [11, 12]. Rather, it presents to potential contributors the internal structure of the SCOTCH software package, and outlines how new partitioning, mapping or ordering methods, whether sequential or parallel, can be added to it.

The rest of the paper is organized as follows. Section 2 presents the API model, the internal data structures, and some coding conventions. Section 3 describes the general architecture of the library, the module naming conventions, and the structure of the computation modules which are the heart of the library. Section 4 outlines the steps to follow to add a new method, in this case a sequential vertex separation method, to the library.

2 API and data structures

2.1 API model

Unlike most partitioning libraries, SCOTCH does not provide an application programmer interface (API) based on single calls. Rather, it is based on the "build-use-destroy" paradigm: prior to performing the actual computations, building routines must be called to fill the data structures on which the core of the library will operate, and destructing routines have to be called afterwards to free the internal arrays which may have been created in the course of computations; this is most true for parallel routines, which have to keep track of halo vertices by means of specific data structures.

2.2 Opaque user data structures

Like for many libraries, API data structures are provided as opaque objects, from which users cannot read or write without resorting to the accessor and mutator routines also provided by the API. In order for compilers to enforce memory alignment constraints when defining and padding these data structures in memory, they are defined, in user include files, as structures containing each a fixed-size data array of double-precision values. The size of the data arrays depends on the architecture for which the library is compiled. To compute and set these sizes properly, a small program, called `dummysizes`, is compiled and executed during the library building process. This solution is efficient, but has some drawbacks. First, it may pose problems when cross-compiling the library for target architectures which have different word sizes than the ones used by the compilation platform. Also, when compiling the parallel version of SCOTCH, the

size of some MPI objects, such as communicator and datatype handles, must be known. The simplest way to do that is to use the `mpicc` compilation command, which provides access to the `mpi.h` include file; however, on some architectures, it requires the resulting program to be run by means of the `mpirun` or the `mpiexec` commands, possibly in batch mode, hence causing compilation problems. This is why the `dummysizes` program has a special compilation line which gives him access to the `mpi.h` include file without requiring to run it through the parallel system of the target machine.

2.3 Integer type

In order for users to be able to handle large graphs on 32-bit systems with extended address space, all integer quantities related to graphs are not directly coded as `int` natural integer types, but instead use a specific type called `Gnum`, which can derive from any integer type at least as large as `int`, such as `int` itself, `long` or even `long long`.

This type is available to users, through the API include file `scotch.h`, as type `SCOTCH_int`. Since this declaration as no equivalent in the Fortran interface, Fortran users have to specify explicitly the size of the integer type of the variables used to interact with the SCOTCH API, by means of `INTEGER*4` or `INTEGER*8` declarations.

2.4 Base value and based accesses

SCOTCH can be called either from C or from Fortran. However, in C, arrays start at index 0, while in Fortran they start at index 1. This is not a problem for accessing the arrays themselves (in both languages, the reference which is passed always points to the first cell of the array), but it impacts how to access an array from indices stored in another array. In this case, the values stored in the indexing array may start from 0 or 1, according to the language used, and if these values are used to access another array, the library must know whether index 1 should refer to its first cell (Fortran style) or to its second cell (C style).

To solve this problem, the SCOTCH API requires that the index base value be passed when declaring data structures which make use of indirect indexing, such as graph structures. This base value, referred to as `baseval` in the SCOTCH coding rules, is used to adjust all internal references to arrays. Let `datatab` be some reference to the first cell of an array as returned from `malloc`, for instance. This array will always be accessed through the `datatax` (for "table access") reference, defined as `datatax` = `datatab` − `baseval`. Hence, `datatax [baseval]` always refers to the first cell of an array, whether indexing starts from 0 or 1. The "`tab`" suffix is reserved to arrays which are never accessed through other index arrays.

2.5 Basic and augmented graph data structures

SCOTCH being a graph-based library, the main data structures it handles are unoriented graphs, declined on many flavors depending on their internal use

(for edge bipartitioning, vertex separation, k-way static mapping, etc). These graphs are classically represented as adjacency lists, with some implementation specificities. Readers interested in these aspects can refer to the user's guides of Scotch [11] and PT-Scotch [12], which describe extensively the implementation of the centralized and distributed graph data types, respectively.

Augmented graph data structures are used, in each computation module, to store the current state of a bipartition, a vertex separator or a static mapping. Basically, a centralized (resp. distributed) augmented graph data structure contains a standard `Graph` (resp. `Dgraph`) structure, plus specific fields depending on the type of augmented graph.

For instance, the centralized bipartition graph `Bgraph` contains, among others, fields which represent the ideal load in part 0 (called `compload0avg`, and which may be different from half of the overall graph load if unequal parts are desired), the current load in part 0 (called `compload0`), an array holding the current part of each vertex (`parttax`), an array holding the indices of all frontier vertices (`frontab`, which accelerates the initialization of FM-like algorithms), etc.

3 Library architecture

The modularity of the API of Scotch is reflected in the modularity of its coding. The source code of Scotch is made of more than 450 files written in C, 410 of which belong to the `libscotch` library itself.

3.1 Global structure

The modular structure of the libScotch library is outlined in Figure 1, page 5. Although it is coded in an imperative language, its design follows the principles of object-oriented programming. A module is a set of files, the functions of which all operate on instances of a common data type, defined by a `typedef struct` construct of the C language. In that respect, the module functions represent the private and public methods of the class the abstract type of which is defined by the structure.

3.2 Naming conventions

Naming conventions strongly reflect our object-oriented coding model. The data structure representing the main data type of a module is named after the name of the abstract type, with a capital first letter, e.g. `Graph` for the centralized graph, `Ordering` for the centralized ordering, `Strat` for the syntactic tree of a parsed strategy, etc.

Types derived from the `Graph` type receive one or more prefix letters, e.g. `Vgraph` is the augmented graph structure used to compute vertex separators (vertex cut), `Bgraph` is used to compute bipartitions (edge cut), `Kgraph` is used to compute k-way static mappings, and `Hgraph` is used to compute halo orderings.

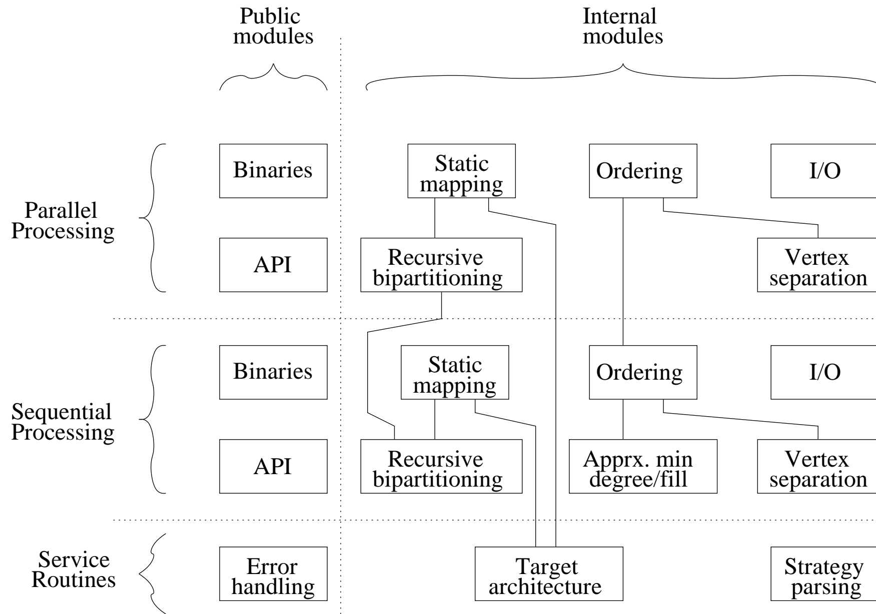

Fig. 1. Sketch of the modular structure of the LIBSCOTCH library. The core capabilities of SCOTCH, that is, static mapping and sparse matrix ordering, are implemented both as sequential and parallel routines, the latter taking advantage of the former when subgraphs are small enough to be handled on a single processing element. Lines represent major dependencies between the main modules.

Similarly, the `Dgraph` type, which defines distributed graphs, is derived into `Vdgraph`, `Bdgraph`, `Kdgraph` and `Hdgraph`.

Every data type of name "`Type`" is defined in a file named `type.h`, with at least a corresponding `type.c` file holding the constructor and destructor routines, among others. All routines are prefixed by the name of the type they apply to. Basic routines have a normalized name, e.g. *type*`Init` for the constructor, *type*`Exit` for the destructor, *type*`Load` and *type*`Save` for I/O, *type*`View` for debug output, etc. In order to save time and improve memory usage, data structures are never allocated dynamically in the constructors themselves. Following the paradigm of passing every variable by reference, every routine is passed a pointer to an instance of the structure on which to operate (analog to the "`this`" reference in object-oriented languages). This pointer is used by the constructor to initialize the structure fields and/or allocate secondary data structures whenever necessary, and by the destructor to free such secondary data structures (but the structure itself is not, as it was not allocated by the constructor).

Callers have therefore full control on memory allocation for data structures: either by allocating them dynamically in the heap, or in the stack as a local

storage, or else statically. This latter option is almost never used in SCOTCH, because global variables prevent routines from being reentrant. Since PT-SCOTCH may make use of threads, and as users of SCOTCH may want to use it in a multi-threaded environment, all of the library routines have been coded so as to be reentrant; the only exceptions concern the strategy parser (depending on the code produced by LEX and YACC), and error handling, for which the name of the calling program is stored as a static variable.

3.3 Consistency checking

As part of our coding standards for quality, every structured type must possess a *type*`Check` routine, to assert the validity of as many data type axioms as possible. For instance, for the `Graph` type, `graphCheck` verifies that for each edge (i, j) there exists an edge (j, i) of same weight, that the graph has no loops, etc. Every violated axiom leads to the output of a specific error message, to track down problems as precisely as possible. These *type*`Check` routines are called at the end of any routine which modifies an instance of the given type. Since these routines can be costly, their calling is conditioned by the definition of a preprocessor debug symbol, one per module. This allows developers to activate only part of the debug routines and save time in other parts of the code.

Large modules are naturally split into several files. Each file name is prefixed by the name of the module, followed by one or more qualifiers describing the contents of the file, separated by underscores, e.g. `dgraph_coarsen.c`, `dgraph_match_sync_coll.c`, etc.

3.4 Structure of computation modules

Computation modules are the modules of the augmented graphs, which contain the expertise of the library. They contain the different methods which can be called through strategy strings. Each method file is identified by a uniform, two-letter suffix: "`fm`" for Fiduccia-Mattheyses (FM) [13] like partitioning algorithms, "`gg`" for greedy graph growing partitioning methods, "`hf`" for halo minimum fill ordering, "`ml`" for the multi-level framework, etc. Hence, the edge and vertex variants of FM methods for centralized graphs can be found in `bdgraph_bipart_fm.c` and `vdgraph_separate_fm.c`, respectively. All of the method files have an associated header file of same name, which contains the prototype of the public method routine, e.g. `vgraphSeparateFm`, and eventually of its private subroutines as well as the definition of internal data types.

For each computation module, a specific file, of method code "`st`", handles the interpretation of the strategy string for the given type of augmented graph. All of such "`st`" method files have the same structure. Basically, they contain two main tables, and an entry point routine. These tables contain all of the data needed by the strategy string parser to build a syntactic tree from strategy strings, and to interpret it at run-time. The first table associates with each single-letter code representing the method (e.g. "`f`" for a FM method), the pointer to the corresponding method routine. The second table lists the names of all the

method parameters, their type, and the offset in the tree node data structure where to write the parsed values.

The entry point routine, when passed pointers to the augmented graph to process and to the root of the strategy syntactic tree, acts according to the type of the tree node. If the node is a method node, it calls the proper method; if the node is a selector node, it recursively calls itself on each of the two sons of the node, recording the partition computed by each, and keeps the best of the two, etc.

All of the methods can take advantage of service routines which operate on the fundamental graph types, e.g. `graphCoarsen` for the sequential multi-level routine, `dgraphInduce` for the parallel nested dissection method, etc.

4 Adding a new method

The LIBSCOTCH has been carefully designed so as to allow external contributors to add their own partitioning or ordering methods. This can be done in few, easy steps.

4.1 Computation modules

There are currently ten types of methods which can be added: sequential k-way graph mapping methods (`kgraph_map_`), sequential graph bipartitioning methods by means of edge separators (`bgraph_bipart_`), sequential graph ordering methods (`hgraph_order_`), sequential graph separation methods by means of vertex separators (`vgraph_separate_`), sequential mesh ordering methods (`hmesh_order_`), sequential mesh separation methods by means of vertex separators (`vmesh_separate_`), parallel k-way graph mapping methods (`kdgraph_map_`), parallel graph bipartitioning methods by means of edge separators (`bdgraph_bipart_`), parallel graph ordering methods (`hdgraph_order_`), and parallel graph separation methods by means of vertex separators (`vdgraph_separate_`). Each of these methods operates on corresponding augmented graph structures. To date, SCOTCH does not comprise any parallel mesh ordering or separation features, nor any mesh partitioning capabilities, whether sequential or parallel.

4.2 Coding the new method

Let us assume that a contributor wants to add a new sequential graph separation routine. This routine will operate on a `Vgraph` structure, and will be stored in files called `vgraph_separate_xy.[ch]`, where `xy` is a yet unassigned two-letter code for the new method. If the implementation of the method requires more than one source file, extended names can be used, such as `vgraph_separate_xy_subname.[ch]`.

In order to ease coding, the files of a simple and already existing method can be used as a pattern for the interface of the new method. Contributors are

encouraged to browse through all existing methods to find the one that looks closest to what they want. In order to understand better the meaning of each of the fields used by augmented graph or mesh structures, contributors can take advantage of the source code of the consistency checking routines, located in files ending with "`_check.c`", e.g. `vgraph_check.c` for the `Vgraph` structure.

Some methods can be passed parameters at run time, from the strategy string parser. These parameters can only be of four types: integer (in fact, a parser `INT`, equivalent to a `Gnum`, so as to be able to store any discrete value as large as the size of the graph), double precision floating-point, enumeration (this `char` type is used to make a choice among a list of single character values, such as "`yn`", which is more readable than giving integer numerical values to method option flags), and strategy (Scotch `Strat` type). Indeed, a method can be passed a sub-strategy of a prescribed type, which can be applied to any augmented graph of this type. For instance, the nested dissection method in `hgraph_order_nd.c` calls the graph separation strategy entry point in `vgraph_separate_st.c` to compute its vertex separators.

The method parameters to be passed from the strategy string must be declared in a specific structure defined in the method header file, e.g. structure `VgraphSeparateXyParam` in file `vgraph_separate_xy.h`.

4.3 Declaring the new method to the parser

Once the new method has been coded, its interface must be known to the parser, so that it can be referred to in strategy strings. All of this is done in the module strategy method file, the name of which always ends in "`_st.[ch]`", e.g. `vgraph_separate_st.[ch]` for the `Vgraph` module. Both the "`.c`" and "`.h`" files must be updated.

In strategy method header file `vgraph_separate_st.h`, a new upper-case `enum` value must be created for the new method in the declaration of the `VgraphSeparateStMethodType` enumerated type, preferably placed in alphabetical order, e.g. `VGRAPHSEPASTMETHXY` (in this only case, "`SEPARATE`" has been shortened in "`SEPA`").

In file `vgraph_separate_st.c`, several updates have to be performed. First, at the beginning of the file, the header file `vgraph_separate_xy.h` of the new method must be added to the list of included method header files, preferably in alphabetical order for the sake of readability.

Then, if the new method has parameters, an instance of the method parameter structure must be created, which will hold the default values for the method. This is in fact a `union` structure, of the following form:

```
static union {
  VgraphSeparateXyParam      param;
  StratNodeMethodData        padding;
} vgraphseparatedefaultxy = { { ..., ..., ... } };
```

where the dots should be replaced by the list of default values of the fields of the `VgraphSeparateXyParam` structure. Note that the size of the `StratNode`

`MethodData` structure, which is used as a generic padding structure, must always be greater than or equal to the size of each of the parameter structures. If the new parameter structure is larger, it is necessary to update the size of the `StratNodeMethodData` type in file `parser.h`. The size of the `StratNodeMethodData` type does not depend directly on the size of the parameter structures (as could have been done by making it an union of all of them) so as to reduce the dependencies between the files of the library. In most cases, the default size is sufficient, and a safety test is added in the beginning of all method routines to ensure it is the case in practice.

Finally, the first two tables of the file must be updated. In the first one, of type `StratMethodTab`, one must add a new line containing the character code used to name the method in strategy strings (which must be chosen among all of the yet unused letters), the pointer to the method routine, and the pointer to the above default parameter structure if it exists (else, a `NULL` pointer must be given). It is essential that the order in which the methods are declared in this table be the same as the one of the `enum` declaration in the strategy header file. Else, the wrong method will be called.

In the second table, of type `StratParamTab`, one line must be added per method parameter, each giving the identifier of the method, the type of the parameter, the name of the parameter in the strategy string, the base address of the default parameter structure, the actual address of the field in the parameter structure (both fields are required because the relative offset of the field with respect to the starting address of the structure cannot be computed at compile-time), and an optional pointer which references either the strategy table to be used to parse the strategy parameter (for strategy parameters) or a string holding all of the values of the character flags (for an enumerated type), this pointer being set to `NULL` for all of the other parameter types (integer and floating point).

4.4 Adding the new method to the Makefile

Of course, in order to be compiled, the new method must be added to the `Makefile` of the `libscotch` source directory. There are several places to update.

First, an entry must be created for the new method source files themselves. The best way to proceed is to search for the one of an already existing method, such as `vgraph_separate_fm`, and to copy it to the right place, preferably in alphabetical order.

Then, the new header file must be added to the dependency list of the module strategy method, e.g. `vgraph_separate_st` for graph separation methods. It is easy to search for the occurrence of string `vgraph_separate_st` to see where this is done.

Finally, the new object file must be added to the contents list of the `libscotch` library file.

Once all of this is done, SCOTCH can be recompiled. It will then able to use the new method within strategy strings.

5 Programming tips

Several programming tips can help programmers of graph algorithms obtain good performance on today's architectures. Here are some of them, which are extensively used in the coding of SCOTCH.

5.1 Removing conditional branches

With the advent of pipelined processors, not breaking execution streams is a prerequisite for good performance. In particular, conditional branches have to be avoided as much as possible. Since processors are now superscalar, it is most often more efficient to perform more computations without conditional branches than to use them; only if branches are heavily biased can branch prediction mechanisms save the day. In the context of graph partitioning, where computations depend on a $\{0, 1\}$ part value (or $\{0, 1, 2\}$ in the case of vertex separators), some arithmetic tricks can help save conditional branches, as illustrated below.

In the two next examples of discrete computations for graph algorithms, the superscalar versions (right) can be three times as fast as their their branching counterparts (left) on classical processors [14]. The first code fragment updates a value positively or negatively according to the part of a vertex. The second computes the number of vertices belonging to parts 0, 1 and 2 from some part array. The tricks below can be adapted to many equivalent situations.

```
if (part[i] == 0)
  value += gain;
else /* (part[i] == 1) */
  value -= gain;
```

```
value += gain * (1 - (part[i] << 1));
```

```
p0 = p1 = p2 = 0;
for (i = 0; i < n; i ++) {
  if (part[i] == 0)
    p0 ++;
  else if (part[i] == 1)
    p1 ++;
  else /* (part[i] == 2) */
    p2 ++;
}
```

```
p1 = p2 = 0;
for (i = 0; i < n; i ++) {
  p1 += part[i] & 1;
  p2 += part[i] & 2;
}
p2 >>= 1;
p0 = n - p1 - p2;
```

5.2 Increasing data locality

Even more critical than good processor usage is good memory usage. It is common for today's high-performance architectures to comprise up to three levels of cache, and misusing them can result in slowdowns of more than one order of magnitude compared to cache-friendly execution. In the case of graph algorithms such as the ones used in the LIBSCOTCH, several tips can help save many memory wait cycles.

The first one regards vertex permutations. In many probabilistic algorithms, such as edge matching, data structure artifacts, such as the order in which candidate vertices are processed, may have a strong impact on the quality of the results. In order to avoid these artifacts, randomization techniques are most often integrated in the algorithms. For instance, the initial queue of vertices to be processed is randomly permuted after having been filled with all vertex indices in order. However, such global permutations negatively impact memory accesses. Since all graph algorithms use adjacency data, when vertices are processed in arbitrary order, their adjacency lists are also fetched in arbitrary order, which may cause many L2/L3 cache and TLB misses (we do not consider page faults, as we expect all data to fit in main memory for performance reasons, all the more in the parallel case). A solution to reduce such L2/L3 cache and TLB misses is to compute cache-friendly permutations: once the queue array has been initialized in order, it is permuted by chunks of a ten of vertices (the size of the chunk is also random to reduce the probability of artifacts). Since the permutation is highly local, we call it a "cache-friendly perturbation" instead. This technique is implemented in all of our graph and mesh coarsening routines.

The second one regards the implementation of hash-tables, acting as data caches, within the FM-like algorithms. Since these algorithms operate by moving vertices close to the current partition boundaries, the ratio of vertices which see their partition and swap gain data being updated at some point of the algorithm is very low, because for most graph families separators are orders of magnitude smaller than graph size [15]. Many FM algorithms already account for this fact by initializing vertex swap gains for boundary vertices only, a technique referred-to as "boundary FM" in the literature [16]. However, full data arrays are reserved for that purpose, such that L2/L3 cache lines corresponding to these arrays are most likely to be underused. To maximize cache line usage, vertex working data is stored in a hash array, indexed by vertex number. Vertices which do not belong to the hash array have not already been touched by the algorithm, and most probably will never be. Once the algorithm completes, only the partition data corresponding to in-hash vertices is updated. Adjacency lists of in-hash vertices are not temporarily gathered to a common space, as no performance gain was experienced when doing so in our prototypes. Indeed, these data already have good L2 locality and are accessed for reading only, such that data re-access is cheap once an adjacency list has been accessed once. Moreover, this gathering of relevant edge data to a smaller working space is already performed when building the band graphs on which our FM algorithms now operate by default [6].

References

1. Pellegrini, F.: Static mapping by dual recursive bipartitioning of process and architecture graphs. In: Proc. SHPCC'94, Knoxville, IEEE (1994) 486–493
2. Pellegrini, F., Roman, J.: SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In: Proc. HPCN'96, Brussels. LNCS 1067 (1996) 493–498

3. Pellegrini, F., Roman, J.: Sparse matrix ordering with SCOTCH. In: Proc. HPCN'97, Vienna. LNCS 1225 (1997) 370–378
4. Pellegrini, F., Roman, J., Amestoy, P.: Hybridizing nested dissection and halo approximate minimum degree for efficient sparse matrix ordering. Concurrency: Practice and Experience **12** (2000) 69–84
5. Chevalier, C., Pellegrini, F.: Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework. In: Proc. Euro-Par'06, Dresden. Volume 4128 of LNCS. (2006) 243–252
6. Pellegrini, F.: A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In: Proc. Euro-Par'07, Rennes. Volume 4641 of LNCS., Springer (2007) 191–200
7. Chevalier, C., Pellegrini, F.: PT-SCOTCH: A tool for efficient parallel graph ordering. Parallel Computing **34** (2008) 318–331
8. Her, J.H., Pellegrini, F.: Efficient and scalable parallel graph partitioning. Parallel Computing (2009) Submitted.
9. CeCILL: ("CEA-CNRS-INRIA Logiciel Libre" free/libre software license) Available from `http://www.cecill.info/licenses.en.html`.
10. GNU: (Lesser General Public License) Available from `http://www.gnu.org/copyleft/lesser.html`.
11. Pellegrini, F.: SCOTCH and LIBSCOTCH 5.1 User's Guide. LaBRI, Université Bordeaux I. (2008) Available from `http://www.labri.fr/~pelegrin/scotch/`.
12. Pellegrini, F.: PT-SCOTCH and LIBSCOTCH 5.1 User's Guide. LaBRI, Université Bordeaux I. (2008)
13. Fiduccia, C.M., Mattheyses, R.M.: A linear-time heuristic for improving network partitions. In: Proc. 19th Design Automation Conference, IEEE (1982) 175–181
14. Pellegrini, F.: Architectures et systèmes des calculateurs parallèles (2008) Class notes. Available from: `http://www.enseirb.fr/~pelegrin/enseignement/enseirb/archsys/cours/c.pdf`.
15. Lipton, R.J., Tarjan, R.E.: A separator theorem for planar graphs. SIAM J. on Appl. Math. **36** (1979) 177–189
16. Hendrickson, B., Leland, R.: The CHACO user's guide – version 2.0. Technical Report SAND95–2344, Sandia National Laboratories (1995)